These notes have not been revisioned and may contain errors

Original author: Erika Lena

Unsupervised Learning

Lecture 1/3/23

What is Machine Learning?

Machine learning is a part of AI that deals with building methods that can learn from dataa.

Methods which, by using data, can improve the performance on some set of tasks (e.g. image recognition).

We have

$$DL \subset ML \subset AI$$

- **AI**: engeenering of intelligent systems, making intelligent systems and programs.
- **ML**: study of algorithms that can learn from data. The ability to learn without being explicitly programmed.

What made ML possible?

- algorithmic development
- data availability
- computational power

What is **unsupervised learning**?

- we learn from data without labels (untagged data)

There exists many level of supervision:

- supervised ML: data tagged by an expert
- reinforcement learning: data tagged, but rules for performance evaluation are given (input: state, output: action, reward)
- semi-supervised learning: some data are tagged, some are not

Supervised learning

- input: tagged data
- output: model (mapping)
- tastes (typologies):
  - regression
  - classification

- model: function that maps input to output

In SL we learn a conditional probability distribution $P(Y|X)$, where X is the input and Y is the output. While in unsupervised learning we want to learn directly properties of the input distribution $P(X)$.

Different properties that we learn from $p(X)$ correspond to different tasks:

- density estimation
- clustering
- dimensionality reduction
- manifold learning
- etc.

MANIFOLD LEARNING

For a certain amount of data, we can think of many possible classifications, but they are not all independent between each other. We can think of a manifold as a surface in the space of the data, that separates the data in different classes.

The dimensionality of the manifold is the number of parameters needed to describe the data. Such that by reducing the number of dimensions to the intrinsic dimensionality of the manifold, we can separate the data in the different classes.

Two questions can be addressed: 1. Which is the dimension of the manifold in which my data lives? (Intrinsic dimension $D \rightarrow d$) 2. Is there a way to projectmy data from $\mathbb{R}^D$ into $\mathbb{R}^d$?

Considerations

- The manifold assumption is usually an approximation
- And the choice of d is **scale dependent**

DENSITY ESTIMATION

How are the data distributed in the space? $p(x)$

CLUSTERING

How can we group the data?

Other tasks from Unsupervised Learning

- Generation of data: Generate new data points x_i that belong to the same distribution p_x (Variational Autoencoders)
- Analysis of time dependent data (Markov State Models)
- Anomaly detection (detect when x_i does not belong to p_x)

Contents

1. Basic notions about Unsupervised Machine Learning.

2. Dimensionality Reduction methods: General theory, classical methods and advance techniques.
3. Intrinsic dimension estimation methods
4. Density Estimation methods: Histograms, kernel density estimation and k-NN. Advanced methods.
5. Clustering: General Theory. Classification of clustering methods. Classical algorithms. Overview about recent developments and new methods. Clustering validation

Dimensionality Reduction Methods

Lecture 14/03/23

PCA

Paper: Pearson 1901

GOAL: Given the dataset $X = \{x_1, ..., x_N\}$, where $x_i \in R^D$, find a representation of y that minimizes the correlations (y_k, y_i).

$x \in R^D$

$y \in R^d$

Step 1: standardize the data

We want to deal with a set of centered data points, so we center the data by subtracting the mean.

$x_k^i = \tilde{x_k^i} - \mu$

where i is the index of the data point and k is the index of the feature. μ is the mean of the feature k, $\mu = \frac{1}{N} \sum_i \tilde{x_k^i}$

We also need to standardize the data, that is to say we need to divide by the standard deviation. $x_k^i = \frac{\tilde{x_k^i} - \mu}{\sigma}$ where σ is the standard deviation of the feature k, $\sigma = \sqrt{\frac{1}{N} \sum_i (\tilde{x_k^i} - \mu)^2}$

Step 2: covariance matrix computation

The aim of this step is to understand how the variables of the input data set are varying from the mean with respect to each other, or in other words, to see if there is any relationship between them. Because sometimes, variables are highly correlated in such a way that they contain redundant information. So, in order to identify these correlations, we compute the covariance matrix.

The covariance matrix is a p × p symmetric matrix (where p is the number of dimensions) that has as entries the covariances associated with all possible pairs of the initial variables. For example, for a 3-dimensional data set with 3 variables x, y, and z, the covariance matrix is a 3×3 data matrix of this from:

e.g.

$$\begin{bmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{bmatrix}$$

where $cov(x,y) = \frac{1}{N}\sum_i (x_i - \mu_x)(y_i - \mu_y)$

In our case, the entries of the covariance matrix are given by:

$C_{mn} = \frac{1}{N}\sum_i (x_m^i)(x_n^i)^T$

because the data is centered, which means the mean is 0.

Since the covariance of a variable with itself is its variance (Cov(a,a)=Var(a)), in the main diagonal (Top left to bottom right) we actually have the variances of each initial variable. And since the covariance is commutative (Cov(a,b)=Cov(b,a)), the entries of the covariance matrix are symmetric with respect to the main diagonal, which means that the upper and the lower triangular portions are equal.

Covariance matrix is symmetric and positive semi-definite by construction.

Why positive semi-definite? It can be shown that for a non-zero vector y, we have that

$$y^T C y = \frac{1}{N}\sum_i (y^T x^i)^2 \geq 0$$

What do the covariances that we have as entries of the matrix tell us about the correlations between the variables?

It's actually the sign of the covariance that matters:

- If positive then: the two variables increase or decrease together (correlated)
- If negative then: one increases when the other decreases (Inversely correlated)

Step 3: eigendecomposition of the covariance matrix

Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. **These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components.** So, the idea is 10-dimensional data gives you 10 principal components, but PCA tries to put maximum possible information (within the initial variables) in the first component, then maximum remaining information in the second and so on.

Tthe principal components are less interpretable and don't have any real meaning since they are constructed as linear combinations of the initial variables.

Geometrically speaking, principal components represent the directions of the data that explain a maximum amount of variance, that is to say, the lines that capture most information of the data. The relationship between variance and information here, is that, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more information it has.

- The first principal component accounts for the largest possible variance in the data set.
- The second principal component is calculated in the same way, with the condition that it is uncorrelated with (i.e., perpendicular to) the first principal component and that it accounts for the next highest variance.

The eigenvectors of the Covariance matrix are actually the directions of the axes where there is the most variance (most information) and that we call Principal Components. And eigenvalues are simply the coefficients attached to eigenvectors, which give the amount of variance carried in each Principal Component.

By ranking your eigenvectors in order of their eigenvalues, highest to lowest, you get the principal components in order of significance.

Step 4: feature vector

The feature vector is simply a matrix that has as columns the eigenvectors of the components that we decide to keep. This makes it the first step towards dimensionality reduction, because if we choose to keep only p eigenvectors (components) out of n, the final data set will have only p dimensions.

Step 5: new data set

In this last step we use the feature vector to transform our data onto the new feature subspace. We do so by dot product between the feature vector and the initial data set. Or, in other words, we multiply the mean-adjusted data set and the matrix of eigenvectors.

$Y = XW$

where Y is a $N \times d$ matrix of the new data set, X is a $N \times D$ matrix of the initial data set, and W is a $D \times d$ matrix of the feature vector, where d is the number of dimensions of the new feature subspace (which is smaller than the original space), and D is the number of dimensions of the original data set.

Derivation of PCA

- We have n points in D dimensions, $x_1, x_2, ..., x_n \in R^D$.
- The component of x_i along the direction p is given by $x_i^T p$.

- The "amount of variation of the dataset along direction p" can be quantified as the empirical variance of the components of the points along p.

$$\frac{1}{n}\sum_{i=1}^{n}(x_i^T p)^2 = \frac{1}{n}\sum_{i=1}^{n} p^T x_i x_i^T p = p^T(\frac{1}{n}\sum_{i=1}^{n} x_i x_i^T)p = p^T C p$$

where C is the covariance matrix of the data.

- Then, finding the direction of maximal variance, amounts to solve the following optimization problem:

$$max_p \;\; p^T C p$$

$$\text{subject to } p^T p = 1$$

- Matrix C can be rewritten as: $XX^T = U\Lambda U^T$ (eigendecomposition)
- Then, the objective function becomes: $p^T U \Lambda U^T p = q^T \Lambda q$ where $q = U^T p$.
- The optimization problem becomes: $max_q \;\; q^T \Lambda q$ subject to $q^T q = 1$.
- Given that the eigenvalues are ordered, the solution is given by $q = [1, 0...0]^T$. And the maximizing p is given by $p = Uq = U[:, 1]$, which is the first column of U, corresponding to the eigenvector associated to the largest eigenvalue.
- The subsequent columns of U represent directions, orthogonal to the previous ones, that maximize the variance of the data.
- The first d columns of U are the eigenvectors associated to the d largest eigenvalues of C. So, in order to obtain our new coordinates:

$$y = U[:, 1:d]^T x$$

where y is a $d \times n$ matrix, $U[:, 1:d]$ is a $D \times d$ matrix and x is a $D \times n$ matrix.

Other clarifications

What we obtained is a projection of the data onto a lower dimensional space, where our projections are given by:

$$y_k^i = \sum_{j=1}^{D} x_j^i u_{kj}$$

where u_{kj} is the j-th component of the k-th eigenvector.

We obtained eigenvectors through the diagonalization of the covariance matrix, with the aim of maximizing the variance of the projected data, which means

that if we look at projected data, the operation we have done corresponds to the maximization of the trace of covariance matrix.

Maximize $Tr(Cov\ \{y_i\})$ in order to preserve as much information as possible in the projected data.

Another constraint is that through eigenvalue-eigenvector decomposition we obtain matrix U, whose vectors are normal:

$$\sum_{j=1}^{D} u_{kj}^2 = 1$$

So, we can write:

$$Tr(Cov\ \{y_i\}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{d} y_k^i y_k^i = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{d}\sum_{l=1}^{D}\sum_{m=1}^{D} u_{kl}\ x_l^i\ u_{km}\ x_m^i$$

Then, since $C_{ml} = \frac{1}{N}\sum_{i=1}^{N} x_m^i\ x_l^i$, we can write:

$$Tr(Cov\ \{y_i\}) = \sum_{k=1}^{d}\sum_{l=1}^{D}\sum_{m=1}^{D} u_{kl}\ u_{km}\ C_{ml}$$

From the constraint we have written above, we can also write the following F function (Lagrange multiplier):

$$F = Tr(Cov\ \{y_i\}) - \sum_k \lambda_k \Big(\Big(\sum_n u_{kn}^2 \Big) - 1 \Big)$$

where λ_k are the Lagrange multipliers, while the second term is the constraint.

Then, we can write the following, by substituting the trace in the F function:

$$F = \sum_{k=1}^{d}\sum_{l,m}^{D} u_{kl}\ u_{km}\ C_{ml} - \sum_k \lambda_k \Big(\Big(\sum_n u_{kn}^2 \Big) - 1 \Big)$$

We will call the above equation our **PCA loss** function.

We aim to **maximize** it, because the solution corresponding to the original constrained optimization is always a saddle point of the Lagrangian function.

This means finding stationary points or equivalently the points in which all partial derivatives are zero.

$$\frac{\partial F}{\partial u_{kn}} = \sum_{k=1}^{d} \frac{\partial}{\partial u_{kn}} \Big[\sum_{l,m}^{D} u_{kl}\ u_{km}\ C_{ml} \Big] - \sum_{k=1}^{d} \frac{\partial}{\partial u_{kn}} \Big[\lambda_k \Big(\Big(\sum_{n} u_{kn}^2 \Big) - 1 \Big) \Big] = 0$$

$$\frac{\partial F}{\partial \lambda_k} = \Big(\sum_{n} u_{kn}^2 \Big) - 1 = 0$$

The second one is easy. Let's decompose the first one in the two parts it is composed by.

Consider the first part and the following options:

$$\begin{cases} \text{if } l \neq n \wedge m \neq n : \dfrac{\partial}{\partial u_{kn}} \Big[u_{kl}\ u_{km}\ C_{ml} \Big] = 0 \\ \text{if } l = n \wedge m \neq n : \dfrac{\partial}{\partial u_{kn}} \Big[u_{kl}\ u_{km}\ C_{ml} \Big] = u_{km}\ C_{mn} \\ \text{if } l \neq n \wedge m = n : \dfrac{\partial}{\partial u_{kn}} \Big[u_{kl}\ u_{km}\ C_{ml} \Big] = u_{kl}\ C_{ln} \\ \text{if } l = n \wedge m = n : \dfrac{\partial}{\partial u_{kn}} \Big[u_{kl}\ u_{km}\ C_{ml} \Big] = 2u_{kl}\ C_{ln} = 2u_{kn}\ C_{nn} \end{cases}$$

Then summarizing, this derivative can be rewritten as:

$$\sum_{m \neq n} u_{km}\ C_{mn} + \sum_{l \neq n} u_{kl}\ C_{ln} + 2u_{kn}\ C_{nn}$$

We can distribute the last term in the first two sums and rewrite the above equation as:

$$\sum_{m} u_{km}\ C_{mn} + \sum_{l} u_{kl}\ C_{ln} = 2 \sum_{l} u_{kl}\ C_{ln}$$

The second term of our derivative is much simpler and it is:

$$\frac{\partial}{\partial u_{kn}} \Big[\lambda_k \Big(\Big(\sum_{n} u_{kn}^2 \Big) - 1 \Big) \Big] = 2\lambda_k u_{kn}$$

Then, we can rewrite our derivative as:

$$\frac{\partial F}{\partial u_{kn}} = \sum_{k} 2 \sum_{l} u_{kl}\ C_{ln} - 2 \sum_{k} \lambda_k u_{kn} = 0$$

$$\sum_k \sum_l u_{kl}\ C_{ln} - \sum_k \lambda_k u_{kn} = 0$$

$$\sum_k \sum_l u_{kl}\ C_{ln} = \sum_k \lambda_k u_{kn}$$

$$\sum_k (UC)_{kn} = \sum_k \lambda_n u_{kn}$$

$$UC = \lambda U$$

Then, we have obtained an "eigenvalue-eigenvector" form and this tells us that if we want to maximize the trace of the covariance matrix of the projected data C_y (indeed the sum above is from $k = 1$ to d), we have to choose the d highest eigenvectors of the covariance matrix of the original data C. qed

Recap:

1. center the data
2. compute the covariance matrix
3. compute the eigenvectors and eigenvalues of the covariance matrix
4. sort the eigenvectors by decreasing eigenvalues and choose k eigenvectors with the largest eigenvalues to form a $D \times k$ dimensional matrix W (where every column represents an eigenvector)
5. compute $Y = W^T X$

How to choose k?

We can plot the eigenvalues in decreasing order, if we can see a clear elbow or a gap, we can choose k as the number of eigenvectors before the elbow (or before the gap). Otherwise we can choose k as the number of eigenvectors that explain at least 90% of the variance, but in this case the decision it more difficult and it needs to be evaluated case by case.

Brief recap:

How it is the covariance matrix of the projected data? **Diagonal**.

Why is it diagonal? Because the projected data is uncorrelated.

Why we want it to be diagonal in first istance? We need a rotation of the space that reduces (cancels) the covariance between the features. Because we want to maintain the smallest possible information contained in our data, discarding everything which can be inferred from the single features that will be mantained, if this is achieved in the covariance between features in the porjected space will be zero. To achieve this we try to diagonalize the covariance matrix C. The matrix W resulting from this process of diagonalization, provides us the desired rotation matrix.

If we choose to keep only the first k eigenvectors, we can reduce the dimensionality of the data from D to k and so we have both a rotation and a projection of the data.

PCA problems

- it is poorly suited for non-linear transformations of the data;
- it is only based on the mean of the data, so it is sensitive to outliers;
- it is not able to capture **invariances** in the data (e.g. rotation of the data);
- it is sensitive to the scaling of the data (it is dependent on the units of measurement of the original variables).

Alternative methods to dimensionality reduction

There exists alternatives focusing for example on preserving distances.

Why distances? Similarities and distances are **much more flexible** than simple correlations. They can be defined in many different ways, and they can be defined in a way that is invariant to many transformations of the data. Each type of data can be associated to its own optimal distance.

MANIFOLD LEARNING

We saw PCA and its problems. Now we will see some alternative methods to dimensionality reduction.

Similarity and Distances

- Clustering tries to separate data "naturally", in such a way that similar elements lay in the same cluster while dissimilar elements belong to a different one
- Similarity S_{ij} is a pairwise function of the features of the elements i and j.
- In terms of space, it can be thought that similar elements are near while dissimilar are far.
- So many times it is useful to talk about "distances between elements" (D_{ij})
- The definition of similarity depends on the nature of the features

Similarity and distance are almost the same concept, but a (metric) distance must accomplish some properties:

1. symmetry: $D_{ij} = D_{ji}$
2. non-negativity: $D_{ij} \geq 0$
3. identity of indiscernibles: $D_{ij} = 0$ if and only if $i = j$
4. triangle inequality: $D_{ij} \leq D_{ik} + D_{kj}$

We can classify distances depending on the nature of the features, which can be:

- quantitative (e.g. height, weight, age, etc.)
- qualitative

Quantitative features

- metric distances:
  - Minkowski distance: $D_{ij} = (\sum_{k=1}^{n} |x_{ik} - x_{jk}|^p)^{1/p}$
    Special cases are:
    - Euclidean distance ($p = 2$)
    - Manhattan distance ($p = 1$)
    - Chebyshev distance ($p \to \infty$)
  - Mahalanobis distance: $D_{ij} = ((x_i - x_j)^T C^{-1} (x_i - x_j))^{1/2}$
    Where C is the covariance matrix of the data. It is a measure of the distance between two points in a multivariate space. It is a generalization of the Euclidean distance.
- non-metric distances:
  - pearson correlation
  - point symmetry distance
  - cosine distance (angle between vectors): $s_{ij} = \frac{\vec{x_i} \cdot \vec{x_j}}{|\vec{x_i}||\vec{x_j}|}$

Qualitative features

- Jaccard similarity index: $S_{ij} = \frac{|A_i \cap A_j|}{|A_i \cup A_j|}$

Where A_i and A_j are the sets of qualitative features of elements i and j respectively.

- Hamming distance: $D_{ij} = n_d = |A_i \cup A_j| - |A_i \cap A_j|$ where n_d is the number of different features between elements i and j.

Lecture 21/3/23

Focusing on preserving distances

Multidimensional scaling

MDS is a technique that tries to preserve the distances between elements in a lower dimensional space. It is a **non-linear dimensionality reduction technique**, while PCA is a linear one.

We want to find a projection of our data in a lower dimensional space that preserves the distances between elements. We can do it by minimizing the difference between the distances in the original space and the distances in the projected space.

- Δ: distance matrix of the original data (in **ambient space**)
- D: distance matrix of the projected data (in **embedded space**)

We want to find y such that $||D - \Delta||$ is minimized.

The input of MDS is a distance matrix Δ and the output is a set of points y_i in a lower dimensional space.

Mathematical setting

To solve the MDS problem, first observe that the solutions are not unique, as any translation or rotation of the projected points will preserve the distances.

We can remove the translation invariance by centering the data. We can do it by subtracting the mean of the data to each point. Which is the same as imposing the following constraint:

$\sum_{i=1}^{N} y_i = 0$

(Assuming Euclidean distance, from now on)

We want the projected points to be as close as possible to the original ones, so we can minimize the following function:

$\sum_{i=1,j=1}^{N} (||y_i - y_j|| - \delta_{ij})^2$

We will call the distance in embedded space $d_{ij} = ||y_i - y_j||$

Ideally we want:

$d_{ij} = \delta_{ij}$

Let's expand this expression:

$d_{ij}^2 = \delta ij^2$
$||y_i - y_j||^2 = \delta ij^2$
$(y_i - y_j)^T(y_i - y_j) = y_i^T y_i + y_j^T y_j - 2y_i^T y_j = \delta ij^2$

Summing over i and j separately, we get:

$\sum_i ||y_i||^2 + \sum_i ||y_j|^2 - 2\sum_i y_i^T y_j = \sum_i \delta_{ij}^2$
$\sum_i ||y_i||^2 + n||y_j|^2 - 0 = \sum_i \delta_{ij}^2$
$\sum_j ||y_i||^2 + \sum_j ||y_j||^2 - 2\sum_i y_i^T y_j = \sum_j \delta_{ij}^2$
$n||y_i||^2 + \sum_j ||y_j|^2 - 0 = \sum_j \delta_{ij}^2$

Where the last term is zero because of the constraint we imposed before.

And denoting by:

$\delta_{\cdot j}^2 = \sum_i \delta_{ij}^2$
$\delta_{i\cdot}^2 = \sum_j \delta_{ij}^2$
$\delta_{\cdot\cdot}^2 = \sum_i \sum_j \delta_{ij}^2$

We can rewrite the equations above as:

$\delta_{\cdot j}^2 = \sum_i ||y_i||^2 + n||y_j|^2$
$\delta_{i\cdot}^2 = \sum_j ||y_j||^2 + n||y_i|^2$
$\delta_{\cdot\cdot}^2 = n\sum_i ||y_i||^2 + n\sum_j ||y_j||^2 = 2n\sum_t ||Y_t||^2$

then

$$\frac{1}{2n}\delta_{\cdot\cdot}^2 = \sum_t ||y_t||^2$$

And so, plugging back, we find:

$$||y_j||^2 = \frac{1}{n}\delta_{\cdot j}^2 - \frac{1}{2n^2}\delta_{\cdot\cdot}^2$$

$$||y_i||^2 = \frac{1}{n}\delta_{i\cdot}^2 - \frac{1}{2n^2}\delta_{\cdot\cdot}^2$$

And finally, we can write:

$$y_i^T y_j = \frac{1}{2}(||y_i||^2 + ||y_j||^2 - \delta_{ij}^2)$$

$$y_i^T y_j = \frac{1}{2}(\frac{1}{n}\delta_{i\cdot}^2 + \frac{1}{n}\delta_{\cdot j}^2 - \frac{1}{n^2}\delta_{\cdot\cdot}^2 - \delta_{ij}^2)$$

We call everything we have on the left g_{ij} and **we have basically written dot product (Gram matrix) in embedded space as a function of the distances in the ambient space.**

- write $G = (g_{ij}) \in R^{N\times N}$, which is Gram matrix for the embedded space.
- $Y = [y_1, y_2, ..., y_N] \in R^{d\times N}$, which is the matrix of the projected (embedded) points.

$$YY^T = G_y$$

Properties of Gram matrix

- if Δ is symmetric (which is the case for distances), then G is also symmetric. Why?

  Because G can be written as:

  $$G = -\frac{1}{2}J\Delta J \quad \text{where } \Delta = (\delta_{ij}^2), J = I_n - \frac{1}{n}11^T$$

  where 1 is a vector of ones.

  Indeed, you can write:

  $$J\Delta J = \Delta - \frac{1}{n}\Delta 11^T - \frac{1}{n}11^T\Delta + \frac{1}{n^2}11^T\Delta 11^T = -2G$$

- All the rows (and columns) of G sum to zero. Due to:

  $J1 = I_n 1 - \frac{1}{n}11^T 1 = 0$

  This also indicates that G must have an eigenvalue equal to 0.

Which must not surprise us, because if we write a matrix of distances, let's say by trying to represent all points in a 2D space, we will have a matrix of distances that is not full rank. Because the embedding of one point (any one) is not free, it is constrained by the distances to the other points.

Remark. The solution of the problem, if it exists, is still not unique. The reason is that any rotation of Y, i.e. YQ for some orthogonal matrix Q, is also a solution, because:

$$(YQ)(YQ)^T = YQQ^TY^T = YY^T = G_y$$

In order for a solution to exist, G must be positive definite. In this case, indeed, we can write:

$$G_y = U\Lambda U^T = U\Lambda^{1/2}\Lambda^{1/2}U^T = U\Lambda^{1/2}\Lambda^{1/2}U^T$$

from which we have the following result.

Theorem. If G is positive semi-definite, and $k \geq r = rank(G)$ (where k is the desired dimensionality of the embedded space), then there exists a solution to the MDS problem.

$$G_y = U\Lambda U^T = (U\Lambda^{1/2})(\Lambda^{1/2}U^T) = (U\Lambda^{1/2})(U(\Lambda^{1/2})^T)^T = (U\Lambda^{1/2})(U(\Lambda^{1/2}))^T = YY^T$$

So that:

$$Y = U\Lambda^{1/2}$$

Where U is the matrix of eigenvectors of G and Λ is the diagonal matrix of eigenvalues of G.

Then, we can write:

$$Y = [\sqrt(\lambda_1)u_1, \sqrt(\lambda_2)u_2, ..., \sqrt(\lambda_k)u_k] \in \mathbb{R}^{N\times k}$$

Remark. If the eigenvalues decay rapidly, we can truncate the columns to obtain an approximate solution.

$$Y_{r0} \approx [\sqrt(\lambda_1)u_1, \sqrt(\lambda_2)u_2, ..., \sqrt(\lambda_r)u_{r0}] \in \mathbb{R}^{N\times r0} \leftarrow \ embedding$$

where $r0$ is the number of eigenvalues that we keep.

Recap of the steps to solve the MDS problem:

1. Compute the Gram matrix $G = -\frac{1}{2}J\Delta J$ from the distance matrix Δ.
2. Compute the eigenvalues and eigenvectors of G.
3. Compute the embedding Y from the eigenvalues and eigenvectors of G.

So what **classical MDS** can be summarized as:

$$\Delta(\rightarrow D) \rightarrow G \rightarrow Y$$

Passing by X instead of Y

$$\Delta \to X \to G \to Y$$

Recap and example

What have we done? We imagined we are given a matrix of squared distances Δ from which we want to find a matrix Y of points in a lower dimensional space such that the distances between the points in Y are as close as possible to the distances in Δ.

Let's say that instead of using matrix Δ, we want to recover X from it and use X in our expression to find G and then Y.

That is to say, we are given:

$$\Delta = \begin{bmatrix} 0 & (x_1 - x_2)^2 & (x_1 - x_3)^2 \\ (x_1 - x_2)^2 & 0 & (x_2 - x_3)^2 \\ \vdots & \vdots & \ddots \end{bmatrix}$$

And we want to recover:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \end{bmatrix}$$

$$\delta_{ij} = (x_i - x_j)^2 = x_i^2 + x_j^2 - 2x_i x_j$$

The strategy used was to convert Δ into matrix B of $x_i x_j$, but before we used the assumption of recovering exact distance, such that:

$||y_i - y_j|| = \delta_{ij}$

And we worked directly with y, now let's see what that means in terms of x.

What we did to the distance $||y_i - y_j||$ (and want to replicate wrt x) was double-centering, by taking:

- the sum of row i of Δ = the sum over column i of Δ, because Δ is symmetric, then:

$$s_i = \sum_j \delta_{ij}^2 = \sum_j (x_i^2 + x_j^2 - 2x_i x_j) = n x_i^2 + \sum_j x_j^2 - 2x_i \sum_j x_j$$

- take the sum over all entries in Δ:

$S = \sum_i s_i = n\sum_i x_i^2 - 2\sum_i x_i \sum_j x_j + \sum_i \sum_j x_j^2 = n\sum_i x_i^2 + n\sum_j x_j^2 - 2(\sum_i x_i)^2 = 2n\sum_i x_i^2 - 2(\sum_i x_i)^2$

If we take $\sum x_i = 0$, that is to say we center our data, then $S = 2n\sum_i x_i^2$.

And then:

$$\delta_{ij}^2 - \frac{1}{n}s_i - \frac{1}{n}s_j + \frac{1}{n^2}S = -2x_i x_j$$

This is the operation of centering Δ, in order to obtain B.

So, to obtain B:

- compute row (or column) sums
- compute sum of all entries (sum of sums)
- apply the formula above to each entry of Δ
- divide by -2

B is the covariance matrix of the centered data and so, if we perform eigenvalue decomposition on it, we obtain exactly the PCA method.

And this is exactly what we did with y, they are just two slightly different ways to do the same thing and to check that:

$$G_y = YY^T = -\frac{1}{2}J\Delta J = \tilde{X}^T\tilde{X} = C_n^T(X^TX)C_n$$

where $\tilde{X}$ is the centered data and C_n is the centering matrix.

Equivalence with PCA

Remark. If the distances are Euclidean, then MDS is equivalent to PCA.

How to prove it?

We should prove that the eigenvalues and eigenvectors of $G = \tilde{X}^T\tilde{X}$ (of size $N \times N$) are the same as those of covariance matrix $C = \tilde{X}\tilde{X}^T$ (of size $D \times D$).

(From now on, we will indicate the centered data with X instead of $\tilde{X}$)

Let's take the following diagonalization of $G = Y^TY = U^T\Lambda U'$.

We have that:

$$(X^TX)v_i = (U^T\Lambda U')v_i = \lambda_i v_i$$

v_i is an eigenvector of G.

Then, multiplying by $\frac{1}{N}X$:

$$\frac{1}{N}X(X^TX)v_i = \frac{1}{N}\lambda_i X v_i$$

where the first part of left term is the covariance matrix $C = \frac{1}{N}XX^T$.

So we have that:

$$CXv_i = \frac{\lambda_i}{N}Xv_i \text{ , which is still an eigenvalue-eigenvector form}$$

we can rewrite it as:

$$Cu_i = \frac{\lambda_i}{N}u_i$$

where $u_i = Xv_i$ is an eigenvector of C.

And $\frac{\lambda_i}{N}$ is an eigenvalue of **covariance matrix** C: from it we can see that the eigenvalues of C are the same as those of G, but scaled by $\frac{1}{N}$. Diagonalizing covariance matrix C or diagonalizing G leads to the same set of eigenvalues that are tighly related.

Same for eigenvectors: $u_i = Xv_i$.

The u_i are orthogonal between them, but not necessarily orthonormal. We can normalize them by dividing by $\sqrt{\lambda_i}$.

$$\hat{u_i} = \frac{u_i}{\sqrt{\lambda_i}} = \frac{Xv_i}{\sqrt{\lambda_i}}$$

Which are the differences between PCA and MDS?

- PCA is a linear method, while MDS is non-linear
- PCA is a projection method, while MDS is an embedding method

In MDS we work with matrix Y of points in the embedded space, while in PCA we work with matrix X of points in the ambient space. So we should get D **eigenvalues and eigenvectors in PCA, while we get** N **in MDS, where** N **is the number of points in the ambient space.**

But in the end, we should have the same number of k top eigenvalues.

[$C = XX^T$ dimensions $D \times D$

$G = X^TX$ dimensions $N \times N$]

MDS

Classical MDS

$s = \sum_{il}(d_{il} - \delta_{il})^2$ equiv. to PCA when we compute Δ using Euclidean distances.

Metric MDS

It is a generalization of classical MDS, where we can use any distance metric.

It uses a numerical optimization method to find the embedding, the configuration of points that minimizes the discrepancy between the original and the reduced distances. It generalizes the optimization procedure to a variety of loss functions and input matrices of known distances with weights and so on.

Metric MDS minimizes the cost function called **stress**, which is a residual sum of squares:

$$s = \sum_{il} \Big(\frac{d_{il} - \delta_{il}}{d_{il}} \Big)^2$$

Here we are interested in relative values of distances, not absolute ones. While PCA is not invariant upon a change in the scale used, this solution is invariant, but we can not solve it analytically. We need to use something like GD or other numerical minimization methods.

Non-metric MDS

It is a generalization of classical MDS, where we can use any distance-like measure (not necessarily a metric, let's call it **dissimilarity**) between items.

$$s = \sqrt{\frac{\sum (f(d) - g(\delta))^2}{\sum g(\delta)^2}}$$

where $f(x)$ is a monotonic function of the dissimilarity d.

We do not minimize directly the difference between distances, but between functions of distances. In this way, we can weight the distances differently, however also the optimization problem becomes more difficult.

What is interesting in classical MDS wrt PCA?

The interesting thing is that we can work with distances and use them for projection, carefully choosing which distances we are interested in.

Still MDS, but using different distances.

ISOMAP

A Global Geometric Framework for Nonlinear Dimensionality Reduction, Tenenbaum et al., 2000

"Unlike classical techniques such as principal component analysis (PCA) and multidimensional scaling (MDS), our approach is capable of discovering the nonlinear degrees of freedom that underlie complex natural observations, such as human handwriting or images of a face under different viewing conditions. In contrast to previous algorithms for nonlinear dimensionality reduction, ours efficiently computes a globally optimal solution, and, for an important class of data manifolds, is guaranteed to converge asymptotically to the true structure."

Isomap is a **non-linear** dimensionality reduction method that uses **geodesic distances** to preserve the intrinsic geometry of the data.

They proposed an approach which combines the classical MDS with the graph theory.

Geodesic distance: the shortest path between two points on a surface.

They argued that only the geodesic distances reflects the true low-dimensional geometry of the manifold.

Given a curve, the distance between two points could be represented by the area under the curve, the problem is that we could have no idea of the path. However, we can approximate the distance in a discrete space where the steps are given by our datapoints.

How to measure distances like they were distances in the manifold?

- we can build a graph using a cut-off graph
- we can build a KNN graph, where each point is connected to its K nearest neighbors, and for the connected nodes the length of the edge (euclidean distance) is used as distance.

Algorithm

1. Determine which points are neighbors of each other on the manifold M,based on the distances $d_x(i, j)$ between pairs of points i and j in the ambient space X.

   Two simple methods are to connect each point to all points within some fixed radius ϵ or to connect each point to its k nearest neighbors.

2. These neighborhood relations are represented as a weighted graph G with nodes V and edges E. The weight w_{ij} of each edge (i, j) is set equal to the distance $d_M(i, j)$ between the two corresponding points i and j on the manifold M.

3. Estimate the geodesic distances $d_M(i, j)$ between all pairs of points as the shortest path distance between them in the graph G.

4. The final step applies classical MDS to the matrix of graph distances $D_G = d_G(i, j)$, constructing an embedding of the data in a low-dimensional Euclidean space Y, which is the one that best preserves the manifold's estimated intrinsic geometry.

Recap

- construct neighborhood graph
- compute shortest path distances
- apply classical MDS to the matrix of graph distances, to find d-dimensional embedding

We can use Floyd-Warshall algorithm to compute the shortest path distances between all pairs of points in the graph.

Kernel methods and other non-linear dimensionality reduction methods

Lecture 28/3/2023

Kernel methods

We are still discussing about different methods to perform dimensionality reduction. We are going to talk about another **non-linear** method called **kernel PCA**.

These methods allow to use **linear classifiers** to solve **non-linear problems**.

What is the **kernel trick**?

Let's immagine we have some data points which are not linearly separable in 2D space. We can project them in a 3D space where they are linearly separable.

$x^i \rightarrow (x_1^i, x_2^i, (x_1^i)^2 + (x_2^i)^2) \quad \forall i$

The question is then: how we can increase the dimensionality of our space without explicitly moving to this higher dimensional space. That is to say: how we can work implicitly with an higher number of dimensions?

The **kernel** is a function of the coordinates in the original space (input space $\mathcal{X}$) which represents the inner product of the coordinates in feature space, $\mathcal{V}$.

$k(x^i, x^j) =< \phi(x^i), \phi(x^j) >_{\mathcal{V}}$

$k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$

is often referred to as **kernel function**.

The kernel reflects a **similarity** between two vectors in the original space, if the vectors are really different (they are orthogonal) then the inner product is zero.

In the example above, the inner product would be:

$k(x^i, x^j) = (x_1^i, x_2^i, (x_1^i)^2 + (x_2^i)^2) \cdot (x_1^j, x_2^j, (x_1^j)^2 + (x_2^j)^2)$

The most simple kernel is the **linear kernel**:

$k(x^i, x^j) = x^i \cdot x^j$

where the ϕ function is the identity function. And so the dimension of the feature space is the same as the dimension of the input space.

$$D_{\mathcal{V}} = D_{\mathcal{X}}$$

The **polynomial kernel** can be:

- $k(x^i, x^j) = (x^i \cdot x^j)^\delta \rightarrow$ include all the polynomials of degree δ, $D_{\mathcal{V}} = \delta D_{\mathcal{X}}$
- $k(x^i, x^j) = (x^i \cdot x^j + 1)^\delta \rightarrow$ include all polynomial up to order δ, $D_{\mathcal{V}} = \binom{\delta + D_{\mathcal{X}}}{\delta}$

Example of second option, with $\delta = 2$ and $D_{\mathcal{X}} = 2$:

$$k_{ij} = (x_1^i x_1^j + x_2^i x_2^j + 1)^2 = (x_1^i x_1^j)^2 + (x_2^i x_2^j)^2 + 1 + 2x_1^i x_1^j + 2x_2^i x_2^j + 2x_1^i x_2^i x_1^j x_2^j$$

which can be rewritten as the dot product between:

$$(1, \sqrt{2}x_1^i, \sqrt{2}x_2^i, (x_1^i)^2, \sqrt{2}x_1^i x_2^i, (x_2^i)^2) \cdot (1, \sqrt{2}x_1^j, \sqrt{2}x_2^j, (x_1^j)^2, \sqrt{2}x_1^j x_2^j, (x_2^j)^2) =$$
$$< \phi^i, \phi^j >$$

The trick is that, instead of computing the dot product in the feature space, we can compute the kernel in the input space, using the first expression:

$$k(x^i, x^j) = (x^i \cdot x^j + 1)^\delta$$

The kernel function enables us to operate in a high-dimensional feature space, without ever computing the coordinates of the data in that space.

The **radial basis function (RBF) kernel** (Gaussian kernel) is:

$$k(x^i, x^j) = \exp(-\frac{||x^i - x^j||^2}{2\sigma^2})$$

where σ is a parameter of the kernel.

e.g.

$$D_{\mathcal{X}} = 2, \frac{1}{2\sigma^2} = 1$$

$k(x^i, x^j) = \exp(-||x^i - x^j||^2) = \exp(-(x_1^i - x_1^j)^2 - (x_2^i - x_2^j)^2) = \exp(-(x_1^i)^2 + 2x_1^i x_1^j - (x_1^j)^2 - (x_2^i)^2 + 2x_2^i x_2^j - (x_2^l)^2) = exp(-||x^i||^2) \cdot exp(-||x^j||^2) \cdot exp(2x^i x^j) = exp(-||x^i||^2) \cdot exp(-||x^j||^2) \cdot \sum_{n=0}^{\infty} \frac{(2x^i x^j)^n}{n!}$

The key restriction is that $\langle \cdot, \cdot \rangle$ must be a proper inner product. This means that the kernel must be symmetric and positive definite.

That is to say k must satisfy the Mercer's condition, which is similar to a result from linear algebra which associates an inner product to any positive-definite matrix.

Condition for k to be a proper kernel: k is positive definite and symmetric.

We previously showed that:

1. classical MDS is equivalent to PCA where Δ is the matrix of Euclidean distances between the data points
2. classical MDS can be written as function of the Gram matrix $G = XX^T$

Then, in order to apply **PCA** over the feature space, it is sufficient to exploit the **Gram matrix** of the kernel function.

The only problem that remains is that the vectors in the feature space won't be centered and we can't center them because we don't know the coordinates in the feature space.

We would like to have: $\tilde{\phi}^i = \phi^i - \frac{1}{N}\sum_k \phi^k$

What we can actually do is rewriting the gram matrix to include this transformation, like we did with gram matrix obtained by using non centered data points.

We can use **double centering** to center it a-posteriori.

$$\tilde{k_{ij}} = \langle \phi^i, \phi^j \rangle - \frac{1}{N}\sum_k \langle \phi^i, \phi^k \rangle - \frac{1}{N}\sum_m \langle \phi^j, \phi^m \rangle + \frac{1}{N^2}\sum_{k,m} \langle \phi^k, \phi^m \rangle$$

Recap - kernel PCA:

1. Decide a kernel
2. Compute the kernel matrix
3. Get the gram matrix by double centering the kernel matrix
4. Compute the eigenvalues and eigenvectors of the gram matrix $\tilde{K}\alpha_k = \lambda_k \alpha_k$, where α_k are the eigenvectors of the gram matrix and λ_k are the eigenvalues
5. Project the data points in the feature space using the eigenvectors of the gram matrix $y_k^i = \sum_{l=1}^{D} x_l^i \alpha_{kl}$

Lecture 30/3/2023

Other **non-linear** methods to perform dimensionality reduction.

Diffusion maps

Diffusion maps is a dimensionality reduction or feature extraction algorithm introduced by Coifman and Lafon which computes a family of embeddings of a

data set into Euclidean space (often low-dimensional) whose coordinates can be computed from the eigenvectors and eigenvalues of a diffusion operator on the data. The Euclidean distance between points in the embedded space is equal to the **"diffusion distance" between probability distributions centered at those points**.

Diffusion maps are a non-linear algorithm, which means that they can be applied to problems where linear methods such as PCA fail. They make use of the concept of random walks to reveal the underlying geometric structure of our dataset. As (imaginary) time progresses, this random walk integrates local features of our data to build a global picture. Using this technique allows one to discover lower dimensional manifolds as well as disconnected clusters in data.

The algorithm is based on diffusion distances. The diffusion distance between two points is the probability that a random walk starting at one point will reach the other point in a given time.

Diffusion maps are part of the family of nonlinear dimensionality reduction methods which focus on discovering the underlying manifold that the data has been sampled from. By integrating local similarities at different scales, diffusion maps give a global description of the data-set. **Compared with other methods, the diffusion map algorithm is robust to noise perturbation and computationally inexpensive.**

Diffusion maps can be defined in 4 steps:

- **connectivity**: dmap exploits the relationship between heat diffusion and random walk markov chain. The basic observation is that if we take a random walk on the data, walking to a nearby data point is more likely than walking to another far away point.

  Let $(X, \mathbb{A}, \mu)$ be a measure space, where X is the dataset and μ represents the distribution of the points on X.

  Based on this, the connectivity k between two data points x and y can be definfed as the probability of walking from x to y in one step of the random walk.

  This probability is usually specified in terms of a kernel function $K(x, y)$, which is a similarity measure between x and y.

  e.g. Gaussian kernel: $k(x, y) = \exp(-\frac{||x-y||^2}{\epsilon})$

  The kernel **constitutes the prior definition of the local geometry of the dataset**. Since a given kernel captures a specific feature of the dataset, its choice should be guided by the application that one has in mind. Which is a major difference wrt PCA et al. where the local geometry is defined by a given measure of distance between points.

  Given (X, k) we can construct a reversible discrete-time Markov chain on X (normalized graph Laplacian construction).

$d(x) = \int_X k(x,y)d\mu(y)$, if our state space is finite, then:

$d(x) = \sum_{y \in X} k(x,y)$

and define

$p(x,y) = \frac{k(x,y)}{d(x)}$

then, we can write our connectivities p_{ij} in a matrix P; if we have only two points:

$$P = \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix}$$

We may recognize this as the **transition matrix of a finite state space Markov chain**. For us, P simply summarizes the probability of moving from one place to another in **one time step**.

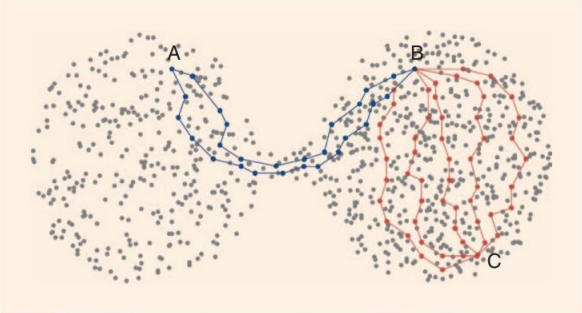
- **diffusion distance** The one just explained was our connectivity measure, we would like to define the notion of **diffusion distance** exploiting the concept of **connectivity**.

  e.g. Even if two points are far away, their diffusion distance can be small if there is a path connecting them through a series of short steps.

  The distance is given by:

$$d_t(x_i, x_j)^2 = \sum_u |p_t(x_i, x_u) - p_t(x_j, x_u)|^2 = \sum_u |P^t_{iu} - P^t_{ju}|^2$$

  $d_t(x_i, x_j)$ is the diffusion distance between x_i and x_j at time t.



[FIG3] **The Euclidean distance and the geodesic distance (shortest path) between points A and B are roughly the same as those between points B and C. However, the diffusion distance between A and B is much larger since it has to go through the bottleneck. Thus, the diffusion distance between two individual points incorporates information of the entire set, implying that A and B reside in different clusters, whereas B and C are in the same cluster. This example is closely related to spectral clustering. For details, see [16] and references therein.**

Intuitively, $d_t(x_i, x_j)$ is small if there is a large number of short paths connecting x_i and x_j in the graph.

t also serves as a scale parameter:

1. Points are closer at a given scale (as specified by $d_t(x_i, x_j)$) if they are highly connected in the graph, therefore emphasizing the concept of a cluster.
2. This distance is robust to noise, since the distance between two points depends on all possible paths of length t between the points.

Why we are interested in the diffusion distance?

Let's start by defining vectors y_i as the collection of all connectivities leading to point x_i:

$$y_i = \begin{bmatrix} p_t(x_1, x_i) \\ p_t(x_2, x_i) \\ \vdots \\ p_t(x_n, x_i) \end{bmatrix}$$

Then, we find that the **euclidean distance** between y_i and y_j is exactly the diffusion distance between x_i and x_j:

$$d_t(x_i, x_j)^2 = ||y_i - y_j||^2$$

Euclidean distance in diffusion space (Y) is the diffusion distance in the original space (X).

How do we compute d_t?

Actually, we do not want to compute diffusion distances, but instead we would like to exploit the previous equality, representing our points as y we will have a Euclidean distance between our points which is the same as the diffusion distance between original data points x.

In order to obtain y, we need P^t, and instead of taking the power of matrix P which is computationally expensive, we can exploit the following equations:

$P = V\Lambda V^{-1}$

$P^t = V\Lambda^t V^{-1}$

And if we let ψ_j denote the j-th column of V (that is to say the j-th eigenvector of P_t), we can write:

$$Y_i = \begin{bmatrix} \lambda_1^t \psi_{1,i} \\ \lambda_2^t \psi_{2,i} \\ \vdots \\ \lambda_n^t \psi_{n,i} \end{bmatrix}$$

where $\psi_{j,i}$ is the i-th component of the j-th eigenvector of P.

!! The eigenvalues of P all lies in $[0, 1]$. In fact, usually only a few eigenvalues are close to 1 and the others are vanishigly small. This means that the diffusion distance is dominated by the largest eigenvalues of P.

As time progresses and t increases these small and intermediate eigenvalues quickly decay. Then, we can drop the terms which are close to zero and approximate Y_i by (for example if we only keep the first three eigenvalues):

$$Y_i \approx \begin{bmatrix} \lambda_1^t \psi_{1,i} \\ \lambda_2^t \psi_{2,i} \\ \lambda_3^t \psi_{3,i} \end{bmatrix}$$

This leaves us with a 3-dimensional reduced space in which euclidean distance is a good approximation of the diffusion distance in the full data space.

N.B. The first eigenvalue is always 1 and the corresponding eigenvector is constant, so it usually discarded; and then, if we keep the first 3, they are actually the 2nd, 3rd and 4th eigenvalues.

Algorithm

Given a dataset X consisting of rows x_i:

1. Compute the kernel matrix K with elements $k(x_i, x_j)$

$$k_{ij} = exp(-\frac{||x_i - x_j||^2}{2\sigma^2})$$

2. Obtain the transitions matrix P by normalizing K:

$$P_{ij} = \frac{k_{ij}}{\sum_{j=1}^{n} k_{ij}}$$

3. Diagonalize P and sort eigenvalues and corresponding eigenvectors in descending order.

4. Truncate the eigenvalues and eigenvectors to the first m largest eigenvalues. The eigenvectors $\psi_1, \psi_2, \ldots, \psi_m$ (with $m \leq n$) span a space of **reduced dimensionality** in which the dataset can be efficiently represented. Multiply the eigenvalues for themselves t times (where t is the number of steps we are interested in) and then multiply the result by the corresponding eigenvectors to obtain the diffusion map coordinates:

$$Y_i = \begin{bmatrix} \lambda_1^t \psi_{1,i} \\ \lambda_2^t \psi_{2,i} \\ \vdots \\ \lambda_m^t \psi_{m,i} \end{bmatrix}$$

To uncover the lower dimensional structure of the data, diffusion map methods build a neighborhood graph on the data based on the distances between nearby points. The graph is then used to construct a Markov chain that describes the probability of transitioning from one point to another. The eigenvectors of the Markov chain are then used to define a lower dimensional representation of the data.

Other non-linear methods

Other **non-linear** methods to perform dimensionality reduction.

Sketch Map

Paper: Simplifying the representation of complex free-energy landscapes using sketch-map, 2011

Recap of what we have already seen, taken from this paper:

The deficiencies of PCA have led researchers to investigate other, nonlinear manifold learning algorithms and in particular locally linear embedding (LLE), Isomap, and diffusion maps.

The first of these, LLE, is a nonlinear approach, which seeks to combine a set of locally linear descriptions in the vicinity of each trajectory frame into a single, unified embedding. It is common knowledge that algorithms like this one are very sensitive to noise. This forces one to question how effective this algorithm can be for molecular trajectories, which are typically very noisy.

The alternative then are **global approaches**, which seek to reproduce all the pairwise distances between the D-dimensional frames by distributing their embeddings in a lower, d-dimensional space. The grandfather of these methods is multidimensional scaling (MDS), which can be solved as an eigenvector problem or by minimization of a stress function. When Euclidean distances are used the eigenvector solution is equivalent to PCA, so approaches involving stress function minimization are often preferred because they are more flexible.

By using a different metric to calculate distances, one can use MDS to fit nonlinear manifolds. For instance, assuming the manifold is isometric with a linear space, one can use the geodesic distance (the distance along the manifold). This idea is the basis of the Isomap algorithm in which geodesic distances are obtained by calculating the length of the shortest path through a fully connected graph that is created by joining the points that are closest together.

Currently the most promising approach for trajectory dimensionality reduction is **diffusion maps**, which can be formulated in a way that makes it resilient to noisy and nonuniformly distributed data. In this approach one defines a weighted graph on the simulation data and then uses the first few eigenvalues of the Laplacian of the manifold as the embedding coordinates. This approach is exciting because

for the systems examined the vectors spanning the low-dimensionality manifold are those in which large barriers to motion make diffusion slow. That said, the method has thus far only been applied to relatively simple systems and not to systems that require one to use accelerated sampling to explore phase space.

One simple way to introduce **nonlinearity** in manifold learning algorithms is to perform distance matching but with the distances transformed or weighted so as to enhance the importance of certain connections.

Sketch-map is essentially a multidimensional scaling, in which the distances in both the high and low dimensional spaces are transformed by a **sigmoid function**, which maps monotonically $\mathbb{R}^+ \to [0, 1)$.

The fact that it is essentially a MDS forces points that are close together in the high-dimensionality space to lie close together in the low-dimensionality space and points that are far apart in the high-dimensionality space to lie far apart. However, because of the form of the sigmoid function, these far apart points can be arbitrarily far apart in the lower dimensionality space.

It is this flexibility that makes the algorithm successful - because it is not trying to accurately reproduce the distances between the far apart data points in the space of lower dimensionality it is free to focus all it's energies on reproducing the connectivity data, which is where there is the evidence that the free energy landscape has a low dimensionality.

The mapping is produced by minimizing the following stress function:

$$L = \Big(\sum_{j \neq i} w_i w_j \Big)^{-1} \sum_{j \neq i} w_i w_j [F(\Delta_{ij}) - f(D_{ij})]^2$$

where w_i is the weight assigned to point i and $\Delta_{ij} = |X_i - X_j|_{(D)}$ and $D_{ij} = |x_i - x_j|_{(d)}$ are the distances in the high and low dimensional spaces, respectively. F and f are general sigmoid functions of the form:

$$s_{\sigma,a,b}(d) = 1 - (1 + (2^{a/b} - 1)(d/\sigma)^a)^{-b/a}$$

where $s_{\sigma,a,b}(\sigma) = 1/2$ and the exponents a and b determine the rate at which the function approaches 0 and 1. The same value of σ is used for both the high and low dimensional spaces; while for a and b we can use different values for each space ($a_{(D)}$, $b_{(D)}$ and $a_{(d)}$, $b_{(d)}$).

$\sigma \to$ scale

$a, b \to$ rate of change

When selecting parameters for the high-dimensionality space sigmoid function F, one is essentially selecting the length scales over which the connectivity data in the high-dimensionality space is interesting.

We should tune σ, a_D and b_D so that for small values of Δ_{ij}, where the histogram of distances resembles that of a full-dimensional multivariate gaussian, we have that $F(\Delta_{ij}) \approx 0$, while for large values of Δ_{ij}, where the histogram of distances resembles that of a uniform distribution, we have that $F(\Delta_{ij}) \approx 1$.

This ensures that, once minimized, points that are close together in the D-dimensional space are mapped close together in the d-dimensional space and vice versa.

Why do we care about **histograms of distances**?

High-dimensionality spaces can often display very nonintuitive properties, which challenge our understanding of distance and proximity. We therefore cannot possibly expect to understand what structures are present simply by visualizing 2D projections. One quantity that can give us some feel as to whether or not it is feasible to represent the data in the lower dimensionality state is the **histogram of pairwise distances**.

The major focus during optimization is the reproduction of distances close to the value of the method's critical parameter, σ, which selects the interesting length scale for the problem. The values of a_D and b_D are far less important and, much like when similar functions are used to calculate continuous versions of coordination numbers, the performance of the method only depends weakly on their values.

The minimization of the equation for L scales quadratically with the number of data points so when fitting a trajectory using sketch-map the first step is to select a small number of landmark frames, which, as detailed elsewhere, can be done either by selecting points at random or by using a farthest point sampling strategy (FPS).

Once the minimization is completed, the projection x of an high-dimensionality point X can be found by minimizing:

$$L(x) = \Big(\sum_{i=1}^{N} w_i \Big)^{-1} \sum_{i=1}^{N} w_i [F(|X - X_i|_{(D)}) - f(x - x_i|_{(d)})]^2$$

where w_i are the weights assigned to the landmark frames X_i, X_i are the landmark points and x_i is its low-dimensional projection.

A global minimum for this quantity can be obtained by calculating the value of L on a grid and then using the lowest-lying point as a start point for a conjugate gradient minimization.

t-SNE

Paper: Visualizing Data using t-SNE

Author: Laurens van der Maaten, Geoffrey Hinton

Year: 2008

t_SNE is a variation of Stochastic Neighbor Embedding (SNE) that is much easier to optimize, and produces significantly better visualizations by reducing the tendency to crowd points together in the center of the map.

tSNE is a relatively simple Machine Learning algorithm which can be covered by the following four equations:

$$p_{j|i} = \frac{\exp\left(-||x_i - x_j||^2/2\sigma_i^2\right)}{\sum_{k\neq i}\exp\left(-||x_i - x_k||^2/2\sigma_i^2\right)}, \qquad p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N} \qquad (1)$$

$$\text{Perplexity} = 2^{-\sum_j p_{j|i}\log_2 p_{j|i}} \qquad (2)$$

$$q_{ij} = \frac{\left(1 + ||y_i - y_j||^2\right)^{-1}}{\sum_{k\neq l}\left(1 + ||y_k - y_l||^2\right)^{-1}} \qquad (3)$$

$$KL(P_i||Q_i) = \sum_i\sum_j p_{j|i}\log\frac{p_{j|i}}{q_{j|i}}, \qquad \frac{\partial KL}{\partial y_i} = 4\sum_j (p_{ij} - q_{ij})(y_i - y_j)\left(1 + ||y_i - y_j||^2\right)^{-1} \qquad (4)$$

Eq. (1) defines the Gaussian probability of observing distances between any two points in the high-dimensional space, which satisfy the symmetry rule.

Eq.(2) introduces the concept of Perplexity as a constraint that determines optimal σ for each sample.

Eq.(3) declares the Student t-distribution for the distances between the pairs of points in the low-dimensional embedding. The heavy tails of the Student t-distribution are here to overcome the Crowding Problem when embedding into low dimensions.

Eq. (4) gives the Kullback-Leibler divergence loss function to project the high-dimensional probability onto the low-dimensional probability, and the analytical form of the gradient to be used in the Gradient Descent optimization.

Problems with t-SNE

1. Dependence on initialization.
2. It is really slow.
3. It does not preserve the global structure of the data: only within cluster distances are meaningful while between cluster similarities are not guaranteed.
4. For $d \geq 3$ it does not work: tSNE can practically only embed into 2 or 3 dimensions. For higher dimensions, the algorithm convergence is really slow if achievable at all.

t-SNE and UMAP make use of probabilistic embeddings.

Other dimensionality reduction methods

UMAP

Paper: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction

Authors: Leland McInnes, John Healy, James Melville

Year: 2020

UMAP is a manifold learning algorithm that is the state of the art in the field.

Differences between UMAP and t-SNE:

1. the way in which we compute the probability of being a neighbor. In UMAP we have:

$$p_{i|j} = e^{-\frac{\Delta_{i,j} - \rho_i}{\sigma_i}}$$

   where ρ is an important parameter that represents the distance from each data point to its first nearest neighbor. This ensures the local connectivity of the manifold. In other words, this gives a locally adaptive exponential kernel for each data point, so the distance metric varies from point to point.

2. It does not normalize nor P, nor Q, that is to say not for either high- or low-dimensional probabilities.

3. UMAP does not use **perplexity**, but it uses instead the number of nearest neighbors k, defined as:

$$k = 2^{\sum_j p_{ij}}$$

4. UMAP uses a slightly different symmetrization of the high dimensional probability:

$$p_{ij} = p_{i|j} + p_{j|i} - p_{i|j}p_{j|i}$$

5. UMAP uses the family of curves $1/(1 + a * y^{2b})$ for modelling distance probabilities in low dimensions, not exactly Student t-distribution but very-very similar, please note that again no normalization is applied:

$$q_{ij} = (1 + a(y_i - y_j)^{2b})^{-1}$$

   a and b are not free hyperparameters, but they depend of the number of nearest neighbors k and the minimum distance ρ.

   This freedom allows UMAP to adapt better to the data that we have.

6. UMAP uses a binary cross-entropy loss function, while tSNE uses the KL-divergence. The cross-entropy loss function is defined as:

$$L = \sum_i \sum_j \left[p_{ij} \; log \frac{p_{ij}}{q_{ij}} + (1 - p_{ij}) \; log \frac{(1 - p_{ij})}{1 - q_{ij}} \right]$$

   This cross entropy has the advantage that it does not take into account only the local probabilities of being neighbors but also the global structure of the data (the rest of the probabilities, which is given by the $(1 - p_{ij})$ term). This is a very important difference between UMAP and tSNE.

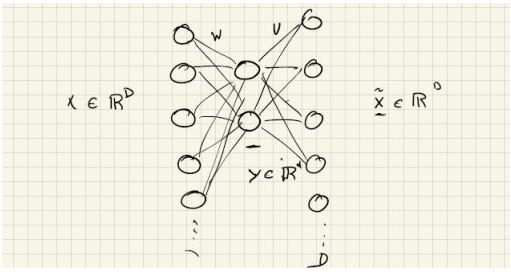
7. UMAP assigns initial low-dimensional coordinates using Graph Laplacian in contrast to random normal initialization used by tSNE. This, however, should make a minor effect for the final low-dimensional representation. However, this should make UMAP less changing from run to run since we do not have a random initialization anymore.

Autoencoders

Introduction to Neural Networks.

An **Autoencoder** is a neural network that is trained to attempt to copy its input to its output. Internally, it has a hidden layer h that describes a code used to represent the input. The network may be viewed as consisting of two parts: an encoder function $h = f(x)$ and a decoder that produces a reconstruction $r = g(h)$.

Representation of the simplest autoencoder we can imagine:



$$y_i = Wx_i$$
$$\tilde{x_i} = Uy_i$$

The loss in this case can be called **reconstruction error** and it is given by:

$L = \sum_i ||x_i - \tilde{x_i}||^2$

We can see how this formulation for autoencoders (using linear activation functions) is formally equivalent to PCA.

$$L = \sum_i ||x_i - \tilde{x_i}||^2 = \sum_i ||x_i - Uy_i||^2 = \sum_i ||x_i - U(Wx_i)||^2$$

We want to show that the goal of minimizing the last quantity is exactly what PCA does.

The proof is in **From Principal Subspaces to Principal Components with Linear Autoencoders** by Elad Plaut.

For the exact equivalence, we must have that activation functions are linear and input must be centered.

Define the architecture of the autoencoder:

- we have an encoder and a decoder;
- weights are obtained by minimizing the reconstruction error;
- once we have the weights, we can use the encoder to obtain the latent representation of our input, which is the original data.
- if parameters become too large and overfitting is possible, then use regularization, by optimizing the following loss function:

  $$L = \sum_i ||x_i - \tilde{x_i}||^2 + \lambda ||W||^p$$

  with $p = 1$ for L1 regularization (Lasso) and $p = 2$ for L2 regularization (Ridge).

- if we assume $y \sim \mathcal{N}(0, \Sigma)$, then we can also use:

$$L = \sum_i ||x_i - \tilde{x_i}||^2 + KL(p(y)||\mathcal{N}(0, \Sigma))$$

Autoencoders

- An autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised manner.
- The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction.
- Recently, the autoencoder concept has become more widely used for learning generative models of data.
- Modern autoencoders have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mapping.
- If the code dimension is larger than the input dimension, an autoencoder tends to learn $g \circ f$ as a identity function.
- An autoencoder whose code dimension is smaller than the input dimension is called **undercomplete**.
- When the encoder and decoder are linear and L is the mean squared error, an undercomplete autoencoder learns to span the same subspace as PCA.
- Undercomplete autoencoders can also fail to learn anything useful if the encoder and decoder are given too much capacity e.g. nonlinearity.
- Regularized autoencoders use a loss function that encourages the model to have other properties besides the ability to copy its input to its output:
  - Sparsity of the representation
  - Robustness to noise or to missing inputs
  - Smallness of the derivative of the representation
- A sparse autoencoder involves a sparsity penalty $\Omega(h)$ on the code layer h, in addition to the reconstruction error:

  $$L(x, g(f(x))) + \Omega(h)$$

  where $g(h)$ is the decoder output and typically we have $h = f(x)$, the encoder output.

Variational Autoencoders

- Variational autoencoders (VAEs) are a type of generative model that is trained using variational inference.

- They add KL-divergence term to the loss function:

$$L(x, g(f(x))) + KL(p(h)||\mathcal{N}(0, I))$$

  where $p(h)$ is the distribution of the latent code h.

  For regression, a standard AE uses MSE(output, target) as the loss function. A VAE uses MSE(output, target) + KL-divergence as the loss function.

- VAEs are similar to regularized autoencoders, but with an additional regularization term that ensures that the latent code has a simple distribution, such as a unit Gaussian. The KL divergence, indeed, punishes latent space values far away from the center; and also every point has a variance that is pushed towards 1.

Lecture 18/4/2023

Intrinsic dimension

There exist different definitions of **intrinsic dimensionality**.

The question is: **how do we choose d?**

e.g.

1. The minimum number of parameters (features) needed to describe the data, with minimal loss of information.
2. Dimension of the manifold in which the data lie. (In manifold learning, we want to estimate the manifold's topological dimension)

Useful paper: Intrinsic Dimension Estimation: Relevant Techniques and a Benchmark Framework

Authors: P. Campadelli, E. Casiraghi, C. Ceruti, and A. Rozza

Brief summary of the paper.

Definition of ID by Bishop:

"a set in D dimensions is said to have an id equal to d if the data lies entirely within a d-dimensional subspace of $\mathbb{R}^D$"

ID estimation techniques proposed in the literature are either founded on different notions of dimension (e.g., fractal dimensions) approximating the topological one or on various techniques aimed at preserving the characteristics of data-neighborhood distributions, which reflect the topology of the underlying manifold. Besides, the estimated id value markedly changes as the scale used to analyze the input dataset changes, and with the number of available points being practically limited, several methods underestimate id when its value is sufficiently high (namely, $id > 10$).

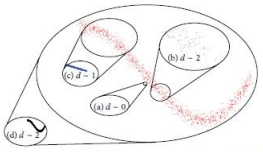


FIGURE 1: At very small scales (a) the dataset seems zero-dimensional; in this example, when the resolution is increased until including all the dataset (b) the id looks larger and seems to equal the embedding space dimension; the same effect happens when it is estimated at noise level (d); the correct id estimate is obtained at an intermediate resolution.

Intrinsic Dimension Estimators

- **Projective id estimators** (eigenvalues methods): since they were designed for exploratory analysis and dimensionality reduction, they often require the dimensionality of subspace of dimension d in which our projections lie. The d we are looking for must essentially be provided as input.

However, their extensions and variants include methodologies to automatically estimate d.

In this category, we have:

  - PCA: when used as an id estimator, the estimate for d is the number of most relevant eigenvectors of the sample covariance matrix, also called **principal components**. And among the deterministic approaches:
    * kernel PCA
    * local PCA
  - MDS:
    * ISOMAP
    * LLE

- **Topological approaches**: they are based on the idea that the intrinsic dimensionality of a dataset is the dimension of the manifold in which the data lie. They are based on the analysis of the data-neighborhood distribution, which reflects the topology of the underlying manifold.
  - **Fractal dimensions estimators**
  - **Nearest-neighbor-based estimators**: they are based on the analysis of the data-neighborhood distribution, which reflects the topology of the underlying manifold.

    They describe data-neighborhoods' distributions as function of d. They usually assume that close points are uniformly drawn from small d-dimensional balls (hyperspheres) $\mathcal{B}_d(x, r)$

    * Trunk's method
    * Pettis's method: An intrinsic dimensionality estimator from Near-Neighbor Information (1979)

      It is notable because it provided a mathematical motivation for the use of nearest-neighbor distances. Indeed, for an i.i.d. sample $P_N \subseteq \mathbb{R}^D$ drawn from a density distribution $p(x)$ in $\mathbb{R}^d$, the following approximation holds:

      $$\frac{k}{N} \approx p(x)V(d)r^d$$

      where k is the number of NN to x within the hypershpere $\mathcal{B}_d(x, r)$, $V(d)$ is the volume of the d-dimensional unit ball, and r is the radius of the hypersphere.

      Since the volume grows as r^d, assuming the density $p(x)$ to be constant, it follows that the number of samples in $\mathcal{B}_d(x, r)$ grows as r^d.

Then, assuming that the samples are locally uniformly distributed, the authors derive the following **id** estimator:

$$\hat{d} = \frac{\bar{r_k}}{k(\bar{r}_{k+1} - \bar{r_k})}$$

where $\bar{r_k}$ is the average distance from each sample point to its k-th NN.

!! remember this approach is strictly correct only if the data are locally uniformly distributed, which means that the density $p(x)$ is constant in the neighborhood of each sample point.

Since the algorithm is limited by the choice of a suitable value of k (number of NN), the authors proposed a variant in which a range of neighborhood sizes $[k_{min}, k_{max}]$ is considered.

However, they also showed that this technique generally yields an understimate of the **id** when d is sufficiently high.

* MLE estimator: it treats the neighbors of each point $p_i \in P_N$ as events in a Poisson process and the distance $r^{(j)}(p_i)$ between the query point p_i and its jth nearest neighbor as the event's arrival time. Since this process depends on d, MLE estimates id by maximizing the log-likelihood of the observed process.

$$\hat{d}(p_i, k) = \Big(\frac{1}{k}\sum_{j=1}^{k} \log \frac{r^{(k+1)}(p_i)}{r^{(j)}(p_i)}\Big)^{-1}$$

where $r^{(j)}(p_i)$ is the distance between p_i and its jth NN.

* Graph-based id estimators

Two-NN id estimator

Summary of paper: **Estimating the intrinsic dimension of datasets by a minimal neighborhood information**

Authors: Facco, Rodriguez 2017

Several approaches of dimensionality reduction work on the assumption that the important content of a dataset belongs to a manifold whose *Intrinsic Dimension (ID)* is much smaller than the ambient space dimension. The aim is to find a low-dimensional representation of the data (with size = ID) that preserves the relevant information.

They propose a new ID estimator using only the distance of the first and second NN of each point in the sample. This extrame minimality enables to reduce the effects of curvature, of density variation, and the resulting computational cost.

The proposed ID estimator is theoretically exact in uniformly distributed datasets, and provides consistent measures in general.

This allows to estimate ID even wen the data lie on a manifold perturbed by a high-dimensional noise.

Intro

Different approaches have been developed to cope with the ID estimation problem. *Projection* techniques look for the best subspaceto project the data by minimizing a projection error or by preserving pairwise distances or local connectivity.

Another point of view is given by **fractal methods**: based on the idea that the volume of a d-dimensional ball of radius r scales as r^d, they count the number of points within a neighborhood of radius r and estimate the rate of growth of this number; these methods in general have the fundamental limitation that in order to obtain an accurate estimation the number of points in the dataset has to be exponentially high wrt the dimension.

A third class of methods is based on the analysis of the data-neighborhood distribution, they try to describe these distributions as functions of the intrinsic dimension d, usually assuming that close points are uniformly drawn from small d-dimensional balls (hyperspheres).

TWO-NN is a new ID-estimator that employs only the distances to the first two nearest neighbors of each point: this minimal choice for the neighborhood size allows to lower the influence of dataset inhomogeneities and to reduce the computational cost.

If the density is approximately constant on the lengthscale defined by the typical distance to the second neighbor it is possible to compute the distribution and the cumulative distribution of the ratio of the second distance to the first one, and it turns out that they are functions of the intrinsic dimension d but not of the density; at this point an equation is obtained that links the theoretic cumulate F to d, and by approximating F with the empirical cumulate obtained on the dataset we are able to estimate the intrinsic dimension.

Density Estimation

Lecture 19/4/23

Density Estimation

In this part, we won't be interested anymore in the properties of our underlying space (e.g. instrinsic number of dimensions), but we will focus on finding a way to directly estimate our data distribution $p(x)$.

Density estimation is the problem of reconstructing the probability density function using a set of given data points. Namely, we observe $X_1, ..., X_n$ and we want to recover the underlying probability density function generating our dataset.

Parametric Density Estimation

The idea is to assume that our data distribution $p(x)$ is a member of a parametric family of distributions $p(x|\theta)$, where θ is a set of parameters. We then need to find the parameters θ that best fit our data. This is done by maximizing the likelihood of our data.

e.g.

Gaussian Mixture Model

We want to obtain $\rho(x) \approx p(x)$ where $\rho(x)$ is provided a Gaussian Mixture Model, through which we try to approximate our true data distribution $p(x)$.

$$\rho(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

where π_k is the mixing coefficient, μ_k is the mean and Σ_k is the covariance matrix of the k-th Gaussian component.

The parametrization is given by $\theta_k = (\pi_k, \mu_k, \Sigma_k)$.

How can we estimate these parameters? Since these are probabilities, we can use the Maximum Likelihood Estimation (MLE) to find the best parameters that fit our data.

$$\mathcal{L} = \prod_{i=1}^{N} \rho(x_i) = \prod_{i=1}^{N} (\sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k))$$

$$log\mathcal{L} = \sum_{i=1}^{N} log(\sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k))$$

Then, to find the best parameters, we need to maximize the log-likelihood function, which can be done using the gradient ascent method. However, if we take the gradient of the log-likelihood function, we will see that it is not possible to find a closed form solution for the parameters. Therefore, we need to use an iterative method to find the best parameters.

$$\frac{\partial log\mathcal{L}}{\partial \theta_k} = \sum_{i=1}^{N} \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)} \frac{\partial log \pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\partial \theta_k}$$

This can be done by using the **Expectation-Maximization (EM) algorithm**.

Non-parametric Density Estimation

The idea is to estimate the density $p(x)$ directly from the data, without making any assumption about the parametric form of $p(x)$.

A non parametric method might for example estimate the density using all observations in a random sample, making all observations in the sample "parameters".

One common approach is kernel smoothing or **kernel density estimation**. The idea is to estimate the density at a point x by averaging the densities of the observations in a neighborhood of x.

Another common approach is to use a **histogram**. The idea is to divide the space into bins and estimate the density in each bin by the proportion of observations falling in that bin.

Histogram Density Estimation

The idea is to divide the space into bins and estimate the density in each bin by the proportion of observations falling in that bin.

$$\hat{p}(x) = \frac{1}{N} \sum_{i=1}^{N} I\Big(\frac{x - x_i}{\Delta}\Big)$$

where I is the indicator function, which is 1 if the argument is true and 0 otherwise and Δ is the width of the bins, N is the number of data points.

Simplifying the above equation, over one interval j, we have:

$$p(x_j) = \frac{n_j}{N\Delta}$$

where n_j is the number of observations in the interval j.

Each data point must of course be assigned to a bin:

$$j(i) = floor(\frac{x_i - x_{min}}{\Delta})$$

The cluster j is retrieved through this simple equation; x_{min} is simply the minimum value of the data points.

The error of the approximation for each bin is proportional to $\frac{\rho(x_j)}{\sqrt{n_j}}$. If we have no points, the error will be of the same magnitude of our estimate.

$$\epsilon_j = \frac{\rho(x_j)}{\sqrt{n_j}}$$

- if Δ is too small, we will have a lot of bins with no data points, which will lead to a lot of zero probabilities.
- while if Δ is too large, all our data points will be in the same bin, which will lead to a single bin with probability 1.

In general Δ can be chosen through cross-validation. Other typical choices are:

- Freedman-Diaconis rule: $\Delta = 2\frac{IQR}{N^{1/3}}$ where IQR is the interquartile range.

Kernel Density Estimation

A kernel is a mathematical function that returns a probability for a given value of a random variable. The kernel effectively smooths or interpolates the probabilities across the range of outcomes for a random variable.

The kernel function weights the contribution of observations from a data sample based on their relationship or distance to a given query sample for which the probability is requested.

A parameter, called the **smoothing parameter or the bandwidth**, controls the scope, or **window of observations**, from the data sample that contributes to estimating the probability for a given sample. As such, kernel density estimation is sometimes referred to as a **Parzen-Rosenblatt window**, or simply a Parzen window, after the developers of the method.

The **kernel density estimator** is given by:

$$\hat{p}(x) = \frac{1}{Nh} \sum_{i=1}^{N} K(x - x_i)$$

where K is the kernel function, and h is the bandwidth.

The most commonly used kernel function is the Gaussian kernel.

In what does this differ from the histogram method? In the histogram method, we have a bin centered around each data point, while in the Parzen-Rosenblatt window method, we have a kernel (a gaussian distribution) centered around each data point.

If we choose a uniform kernel, we will have the same result as the histogram method.

- uniform kernel: $K(x) = \frac{1}{2}\chi(x, x_i, h)$, with $\chi(x, x_i, h) = 1$ if $|x - x_i| \leq h$ and 0 otherwise.
- gaussian kernel: $K(x) = \frac{1}{\sqrt{2\pi} \cdot h} e^{-\frac{1}{2}\left(\frac{x - x_i}{h}\right)^2}$

h is the bandwidth, which is the width of the kernel.

Rule-of-thumb bandwidth selection

If Gaussian basis functions are used to approximate univariate data and the underlying density being estimated is Gaussian, then the optimal choice for h is:

$$h = \left(\frac{4\hat{\sigma}^5}{3N}\right)^{1/5}$$

where $\hat{\sigma}$ is the sample standard deviation.

Sometimes $\hat{\sigma}$ is replaced by the interquartile range (IQR) of the data.

Another modification which improves the model is reducing coefficient from ~ 1.06 to 0.9:

$$h = 0.9 \cdot min\left(\hat{\sigma}, \frac{IQR}{1.34}\right) \cdot N^{-1/5}$$

This approximation is called *normal distribution approximation* or *Silverman's rule of thumb*.

Of course bins size can be chosen as well by using Cross Validation.

Lecture 26/04/2023

K-Nearest Neighbor Density Estimation

Another important way in which to **estimate density** is **K-Nearest Neighbor density estimation**, it is a non-parametric method.

It is really similar to what is done for k-NN in supervised learning. The idea is to estimate the density at a point x by averaging the densities of the k nearest neighbors of x.

$$p(x) = \frac{k}{V_k} = \frac{k}{V^d \cdot r_k^d}$$

where V_k is the "hyper"-volume occupied by the k nearest neighbors of x.

We need to add also a normalization factor, in order for our density to be a proper probability density function.

$$p(x) = \frac{k}{N \cdot V_k} = \frac{k}{N \cdot V^d \cdot r_k^d}$$

where N is the number of observations in our dataset.

The variance of the density estimator is inversely proportional to the number of neighbors k, under the assumption that the density is constant in the neighborhood of x.

Clustering

Lecture 26/04/23

Clustering

We will dedicate the rest of the course to this topic, which is very important.

It is not the only thing of unsupervised learning. It is the unsupervised analogous of classification in supervised context.

Clustering is not classification

Many times we are interested in performing clustering according to some specific features we are interested in. The idea is that for doing **feature selection** we need to have some **prior knowledge** about the data, which is something we do not have in unsupervised learning, a **ground truth (classification)**.

What we can do in order to mirror some sort of classification?

1. variable ranking
   - compare each feature with the GT and quantify
   - sort by correlation, choose those which are more correlated
     - linear correlation
       How to compute linear correlation?

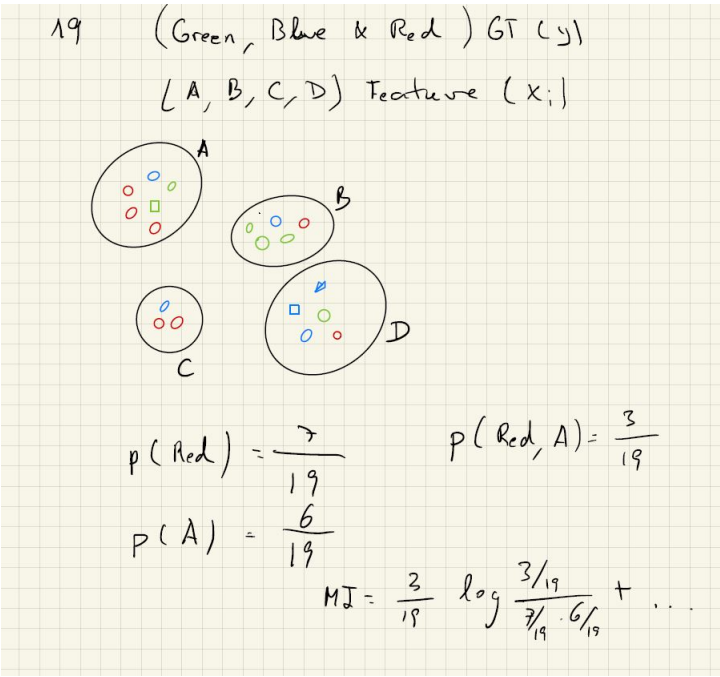
$$R(i) = \frac{\sum_k (x_i^k - \bar{x_i}^k)(y^k - \bar{y})}{\sqrt{(\sum_k (x_i^k - \bar{x_i}^k)^2 \sum_k (y^k - \bar{y})^2)}}$$

$$R(i)_{X,Y} = \frac{E([X - \mu_X][Y - \mu_Y])}{\sigma_X \sigma_Y}$$

       where μ_X and μ_Y are the means of X and Y respectively, and σ_X and σ_Y are the standard deviations of X and Y respectively. This is the **Pearson correlation coefficient**. Linear correlation coefficient between our variable x_i and the ground thruth y.
     - then there is the case in which our GT variable is categorical. In this case we can express correlation as a **mutual information**.

$$MI(i) = \int \int p(x_i, y) \log \left(\frac{p(x_i, y)}{p(x_i)p(y)} \right) dx_i dy$$

where $p(x_i, y)$ is the joint probability distribution of x_i and y, and $p(x_i)$ and $p(y)$ are the marginal probability distributions of x_i and y respectively.
Let's make an example:
We have 19 datapoints and (Green, Blue and Red) is our GT(y) (A, B, C, D) features (x_i)



Variable rankings (Drawbacks)

There are important issues to consider when performing variable ranking. e.g. variables are not supposed to be independent, it could happen that we consider the same variable two times, but with different names, or that we consider two variables which are highly correlated.

1. Not clear how to deal with **redundant variables**
2. A feature may appear to be useless if considered alone, but it may be useful in combination with other features. (e.g. XOR problem) And if we select features as we did before, we will not be able to select these kind of features.

Another way to select variables, which tries to solve this:

Subset selection

1. Explore **subsets of features** and
2. Rank them

However this is **computationally expensive** and it is not really used in practice.

CLUSTERING

Instead of selecting features, we can try to **group** data points according to some **similarity**.

Clustering is the process of **grouping** a set of **objects** in such a way that **objects** in the **same group** (called a **cluster**) are more **similar** (in some sense) to each other than to those in other groups (clusters).

Cluster: a collection of data objects that are similar to one another within the same cluster and are dissimilar to the objects in other clusters.

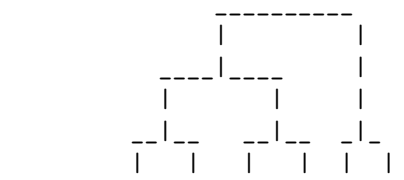
It is really hard to define (also mathematically) what a cluster is, because it is a **subjective** concept. It depends on the **context** and on the **application**.

We can classify according to the **output** of the clustering algorithm.

What kind of output we can have?

- **Flat clustering**: one to one mapping between data points and clusters.
- **Fuzzy clustering**: one to many mapping between data points and clusters. $\overrightarrow{\mu(i)}$ degree of membership of x_i to cluster C_j. It is not exactly a probability, but however we must have that: $\sum_j \mu(i)_j = 1$.
- **Hierarchical clustering**: we have a tree structure (not a sigle classification), which is a **dendrogram** (a binary tree). We can cut the tree at different levels, obtaining different clusterings. We can also obtain a **flat clustering** by cutting the tree at a certain level (at higher level, we would have less clusters made of more data points, at lower level we would have more clusters made of less data points).

Hierarchical clustering



We will see methods that are instances of these three types of clustering.

K-means clustering

It is based on two concepts: - maximizing intercluster dissimilarity - minimizing intracluster dissimilarity

$$L(z) = \sum_{l}^{k} \sum_{i}^{N} \delta(z_i, l) ||x_i - C_l||^2$$

where z_i is the cluster assignment of x_i, C_l is the centroid of cluster l, and $\delta(z_i, l)$ is the Kronecker delta function, which is 1 if $z_i = l$ and 0 otherwise.

$$C_l = \frac{\sum_i^N \delta(z^i, l) x^i}{\sum_i^N \delta(z^i, l)}$$

- z is the **clustering vector**
- z^i is the cluster element i belongs to

We want to **minimize** loss function $L(z)$.

We can try all the possible combinations and measure the loss, but this is quite unfeasible of course.

We can use instead an **iterative algorithm**.

K-means algorithm

1. Pick k random points as initial centroids
2. Assign each point to the closest centroid $\rightarrow z^i = argmin_l ||x^i - C_l||^2$
3. Update each centroid to the **mean of the points assigned to it**
4. Repeat 2 and 3 until convergence (convergence means that the centroids do not change anymore, that is to say no assignment changes anymore)

Lecture 2/5/23

K-means: a Flat Clustering algorithm

- K-means attempts to minimize the intracluster distance while maximizing the intercluster distance.
- it is based on the concept of cluster centroid
- it is still widely used
- it can be parallelized and linearized
- ISSUE: the user must provide k

Loss (objective) function:

$$L(z) = \sum_{l}^{k} \sum_{i}^{N} \delta(z_i, l) ||\overrightarrow{x_i} - \overrightarrow{C_l}||^2$$

where z_i is the cluster assignment of x_i, C_l is the centroid of cluster l, and $\delta(z_i, l)$ is the Kronecker delta function, which is 1 if $z_i = l$ and 0 otherwise.

This function is really hard to be optimized, so we use an **iterative algorithm**.

1. Initialize k centroids randomly
2. Assign each point to the closest centroid $\rightarrow z^i = argmin_l ||x^i - C_l||^2$
3. Update each centroid to the **mean of the points assigned to it**
4. Repeat 2 and 3 until convergence (convergence means that the centroids do not change anymore, that is to say no assignment changes anymore)

Better initialization: k-means ++

1. Choose the first cluster center at random
2. Repeat until all k centers have been found
   - For each point x_i, compute $D(x_i)$, the distance between x_i and the nearest center that has already been chosen
   - Choose one new data point at random as a new center, using a weighted probability distribution where a point x_i is chosen with probability proportional to $D(x_i)^2$
3. Proceed with standard k-means algorithm

This is a better initialization, because it tries to avoid bad initialization, which can lead to bad results.

Convergence will occur faster than with random initialization.

K-means weaknesses

- Initialization sensitive (local optimization) $\rightarrow$ k-means ++

- which k-employing → scree test
- sensitive to outliers → use of medians instead of means (k-medoids)
- employ Euclidean distance → use of other distances (e.g. Mahalanobis distance)
- only for spherical clusters → kernel k-means

K-medoids

What does it have in common with k-means?

They work the same except for the way in which the choose centroids.

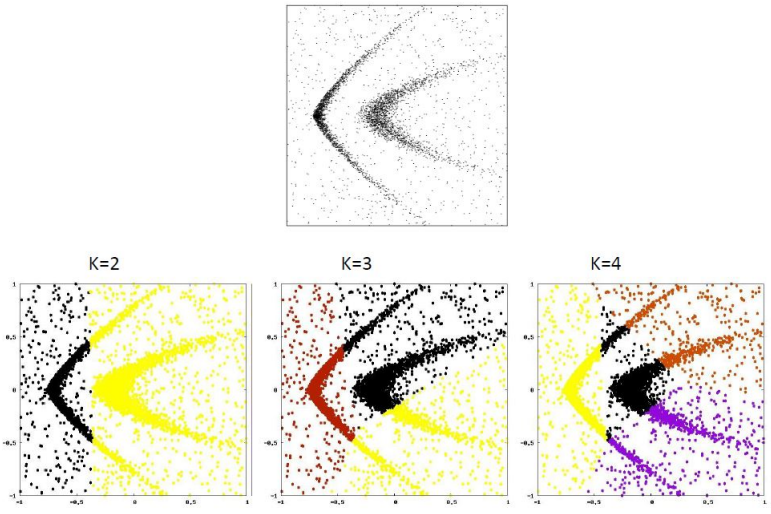
Which is the difference?

In contrast to the k-means algorithm, k-medoids chooses **actual data points** as centers (medoids or exemplars), and thereby allows for greater interpretability of the cluster centers than in k-means, where the center of a cluster is not necessarily one of the input data points (it is the average between the points in the cluster). Furthermore, **k-medoids can be used with arbitrary dissimilarity measures**, whereas k-means generally requires Euclidean distance for efficient solutions. Because k-medoids minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances, it is **more robust** to noise and outliers than k-means.

Pro

- it can be used with any distance metric
- it is less sensitive to outliers

We still have problems, eg. with situations like the one in figure below, where we do not have convex clusters.



Kernel k-means

How to deal with non-linearly separable data?

We can use a **kernel**, in order to map the data into a higher dimensional space, in which the data will be linearly separable.

We want to write **k-means** as a function of the **inner product**.

We start from distance in feature space, seeing that they can be computed as function of the inner products.

$$||\overrightarrow{\Phi}^i - \overrightarrow{\Phi}^l||^2 = \overrightarrow{\Phi}^i\overrightarrow{\Phi}^i + \overrightarrow{\Phi}^l\overrightarrow{\Phi}^l - 2\overrightarrow{\Phi}^i\overrightarrow{\Phi}^l = K_{ii} + K_{ll} - 2K_{il}$$

The loss function was:

$L = \sum_l^k \sum_i^N \delta(z^i, l)||\overrightarrow{x}^i - \overrightarrow{C}_l||^2$

$\overrightarrow{C}^l = \dfrac{\sum_i^N \delta(z^i, l)\overrightarrow{x}^i}{\sum \delta(z^i, l)}$

The **distance** is the following, in which we substitute the expression of $\overrightarrow{C}_l$:

$$||\overrightarrow{x}^i - \overrightarrow{C}_l||^2 = (\overrightarrow{x}^i - \overrightarrow{C}_l)(\overrightarrow{x}^i - \overrightarrow{C}_l) = \overrightarrow{x}^i \cdot \overrightarrow{x}^i - 2\overrightarrow{x}^i\overrightarrow{C}_l + \overrightarrow{C}_l \cdot \overrightarrow{C}_l =$$

$$= \overrightarrow{x}^i\overrightarrow{x}^i - 2\frac{\sum_n \delta(z_n, l)\overrightarrow{x}^i\overrightarrow{x}^n}{\sum_n \delta(z^n, l)} + \frac{(\sum_m \delta(z^m, l)\overrightarrow{x}^m) \cdot (\sum_n \delta(z^n, l)\overrightarrow{x}^n)}{\sum_m \delta(z^m, l)\sum_n \delta(z^n, l)}$$

Then, we can write:

$$||\overrightarrow{\Phi}^i - \overrightarrow{C}_l|| = K_{ii} - 2\frac{\sum_n \delta(z_n, l)K_{in}}{\sum_n \delta(z_6, l)} + \frac{\sum_{n,m} \delta(z^n, l)\delta(z^m, l)K_{nm}}{(\sum_n \delta(z^n, l)^2)}$$

If we want to compute the kmeans in the kernel space, we need to apply this last formula (definition).

1. Compute kernel matrix (whatever kernel we want)
2. Assign random cluster assignments
3. Start iterating this equation: $z^i = argmin_l(d_{il}^2)$ where z^i is the cluster assignment of x^i and d_{il}^2 is the distance between x^i and C_l in the kernel space.

If we know the data, we can decide to use a specific kernel.

Fuzzy k-means (or soft k-means, or fuzzy c-means)

Fuzzy clustering: one to many mapping between data points and clusters. $\overrightarrow{\mu}_j(i)$ degree of membership of x_i to cluster C_j. It is not exactly a probability, but however we must have that: $\sum_j \mu(i)_j = 1$.

Now we will express z^i as a vector of probabilities, which is the degree of membership of x^i to each cluster.

$\overrightarrow{\mu}_l^i = \delta(z^i, l)$

e.g.

if $z_i = 3 \rightarrow \overrightarrow{\mu}^i = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$

if $z_j = 5 \rightarrow \overrightarrow{\mu}^j = (0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$

We just want to have $\sum \mu_l^i = 1$.

We are going to see how our method changes.

First thing: how to compute now the loss function?

$L(z) \rightarrow$ k_means

$L(U) \rightarrow$ fuzzy k_means

$L(U) = \sum_l^k \sum_i^N \mu_l^i ||\overrightarrow{X}^i - \overrightarrow{C}_l||^2$

where μ_l^i is the degree of membership of x^i to cluster C_l.

$$C_l = \frac{\sum_i^N \mu_l^i \overrightarrow{X}^i}{\sum_i^N \mu_l^i}$$

Until here the parallelism is almost one-to-one with k-means.

From now on we will have some differences.

We want to find matrix U

1. Initialize U randomly
2. If we have a fixed U, we can compute the centroids as before $C_l = \frac{\sum_i^N (\mu_l^i)^f \overrightarrow{X}^i}{\sum_i^N (\mu_l^i)^f}$

f is a parameter which symbolizes the **fuzziness** of the algorithm. If $f = 1$ we have the same as before, if $f > 1$ we have a fuzzier clustering, if $f < 1$ we have a less fuzzy clustering.

3. update membership $\mu_i^l = \frac{1}{\sum_n^k \left(\frac{||\overrightarrow{x}^i - \overrightarrow{C}_l||}{||\overrightarrow{x}^i - \overrightarrow{C}_n||} \right)^{\frac{2}{f-1}}}$

This formula has been derived to be equivalent to the argmin formula, if we go for $f \to \infty$.

4. Check the convergence on U (if it does not change anymore, we have convergence) We have to define a criterion for checking this.

This algorithm has the same problems as k-means:

- it depends on initialization
- it is not well suited for non-convex clusters
- it is not easy to define k

One thing that can improve convergence is to perform simple **k-means** and then use the result as initialization for **fuzzy k-means**.

However, in this case it is even more difficult to check that we converged to a global minimum.

Hierachical clustering

Lecture 9/5/23

RECAP:

We have seen how to learn properties of the underlying probability distribution function that we assume generated our data.

Among the methods we have seen those that are connected with manifold learning study which are:

- projection methods and
- intrinsic dimension estimation

Then we saw how to **estimate density** through: **histograms**, **KDE** and **K-NN**.

Finally, we saw **grouping** methods, which are the unsupervised analogous of classification in supervised context.

- flat clustering (kmeans, k-medoids, kernel kmeans, kmeans++)
- fuzzy clustering (fuzzy c-means)
- hierarchical clustering

So, today we are going to dive into methods for **Hierarchical clustering**.

How can we obtain the tree structure?

- we consider all our points into a unique cluster
- we start making partitions
- we continue until we have only one point per cluster

This is what is called the **Divisive clustering**.

Another option in which we can build a tree, is the **Agglomerative clustering**. We start considering each single data point as a cluster and then we merge them until we have only one cluster.

What we obtain in both cases are not usual trees, but a particular type which is called **dendrogram**.

Agglomerative clustering is the most used method, because it has a practical advantage: it is more efficient than **Divisive clustering**.

Divisive clustering has to consider all the possible partitions, so it is computationally expensive. We have $2^{N-1} - 1$ partitions. Divisive clustering is more complex as compared to agglomerative clustering.

Agglomerative clustering algorithm:

1. compute the distance matrix between all the points
2. consider each data point as a cluster
3. merge the two closest clusters
4. update the distance matrix
5. repeat steps 3 and 4 until only one cluster remains

How to compute the distance between clusters?

- **Single Linkage Clustering** → minimum distance among two elements in two different clusters

  How do we update the distance matrix? We simply consider the minimum distance between the two clusters, looking at all data points in the two clusters.

  This criterion has some problems and some advantages.

  Advantage: it works with arbitrary shaped clusters.

  Disadvantage: it is sensitive to noise. Why? Well if we consider pairs distances, by adding spurios points, results can change quite significantly.

- **Complete Linkage Clustering** → maximum distance among two elements in two different clusters.

  Once distances are computed, we can merge the two closest clusters as before.

  This second method generates **balanced clusters** and it is **not sensitive to noise**.

  However the main disadvantage is that it forces **balanced structure**, which is not always the case.

Lecture 16/5/23

Recap

Agglomerative clustering

We started talking about hierarchical clustering, which is a method to build a tree structure that represents the data.

- Hierarchical clustering
  - Divisive

- Agglomerative: general algorithm → merge at each step the two closest clusters → DENDOGRAM What is determinant is how we compute the distance between clusters.
  - **Single Linkage Clustering**: distance between two clusters is the minimum distance between two elements in the two clusters $d_{AB} = min[d_{ij}|i \in A, j \in B]$
  - **Complete Linkage Clustering**: distance between two clusters is the maximum distance between two elements in the two clusters $d_{AB} = max[d_{ij}|i \in A, j \in B]$
  - **Average Linkage Clustering**: distance between two clusters is the average distance between two elements in the two clusters **group average linkage** $d_{AB} = \frac{1}{|A||B|} \sum_{i \in A} \sum_{j \in B} d_{ij}$

Other options are:

- **Group average linkage**: it tends to produce compact clusters, and balanced clusters. It is a bit like a mild version of complete linkage clustering.
- **Centroid linkage**
  - compute centroids of clusters: like in k-means
  - compute distance between centroids, which provides the distance between clusters

  This kind of distance has the same advantages and disadvantages of the group average linkage. It is simpler to be computed, but it is not so used in particular because, when the clusters become larger, the distance between the centroids could be not so representative of the distance between the clusters.
- **Ward's linkage**

  This is probably the most used method to do agglomerative hierarchical clustering. It is a bit more complex to be computed, but it is more robust to noise and it is more efficient.

$$d^2_{AB} = \frac{2|A||B|}{|A| + |B|} |C_A - C_B|^2$$

where C_A is the centroid of cluster A, C_B is the centroid of cluster B.

It can be shown that, for disjoint clusters A, B and C, the following holds:

$$d_{A \cup B/C} = \frac{|C_A| + |C_C|}{|C_A| + |C_B| + |C_C|} \cdot d_{AC} + \frac{|C_B| + |C_C|}{|C_A| + |C_B| + |C_C|} \cdot d_{BC} - \frac{|C_C|}{|C_A| + |C_B| + |C_C|} \cdot d_{AB}$$

This means we want to move from the three clusters to two, by merging A and B, and we want to minimize the increase of the variance. Written like this the update of the distance matrix becomes easy.

When we don't have much noise we can go for methods like **Ward's linkage** or **Single Linkage Clustering**.

When we have a lot of noise we have to search in bibliography for the best method to use or use **Ward's Method**.

Divisive clustering

DIANA $\rightarrow$ Divisive Analysis

Why divisive clustering is not so used?

The problem is that we have to consider all the possible partitions, so it is more computationally expensive. $2^{N-1} - 1$ partitions.

Let's say we have a matrix of dissimilarities

	A	B	C	D	E
A	0	2	6	10	9
B	2	0	5	9	8
C	6	5	0	4	5
D	10	9	4	0	3
E	9	8	5	3	0

1. We can compute the **average dissimilarity** to other elements: let's say we have:
   - for a 6,75 = (2+6+10+9)/4
   - for b 6,0 = (2+5+9+8)/4
   - for c 5,0 = (6+5+4+5)/4
   - for d 6,5 = (10+9+4+3)/4
   - for e 6,25 = (9+8+5+3)/4
2. choose the element with the highest average dissimilarity, a in this case, to nucleate a new cluster
3. compute the average dissimilarity between a and the other elements (b, c, d, e)
   - for b, c, d, e we must compute the average dissimilarity with the new cluster a and then with the other elements

     e.g. for b they would be c, d, e

     AN stands for average dissimilarity with the new cluster a and AP stands for average dissimilarity with the other elements

	AN	AP	Δ = AP - AN
b	2	7,33	5,33
c	6	4,67	-1,33
d	10	5,33	-4,67
e	9	5,33	-3,67

4. consider elements with positive Δ as members of the new cluster
5. repeat steps 3 and 4 until no positive Δ is found

After first iteration, we would have clusters {a,b} ad {c,d,e}. Then we have to repeat, starting by computing the average dissimilarity between {a,b} and {c,d,e}. We would have:

	AN	AP	Δ = AP - AN
c	5,5	4,5	-1
d	9,5	3,5	-6
e	8,5	4,0	-4,5

To continue our partitions: we will consider each cluster obtained and:

6. consider the cluster with the highest pair dissimilarity for performing the next partition
7. repeat steps 3 to 6 until no positive Δ is found, stop when each cluster contains only one element
   - choose the cluster with the highest pair dissimilarity;
   - nucleate the new cluster around the point with the highest average dissimilarity to the rest of the points in the cluster.

Spectral clustering

Lecture 17/5/23

Until now, we have seen classical methods for clustering, which are based on the idea of **distance** between points.

Now we will see modern clustering algorithms.

Spectral clustering

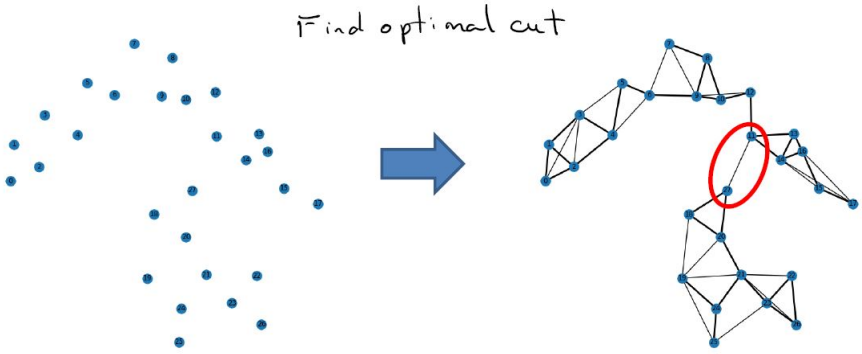
We transform the points into a graph and then we apply graph clustering algorithms. Try to find the cut in the graph which is somehow *optimal*.

Clusters: the not connected components of the graph.

We deal with **undirected weighted graphs** $G(E, V)$

Edges are defined as: $E\{(i, l), S_{il} \geq 0\}$, S_{il} is a certain weight which expresses similarity between the points i and l. We can put all of these in a similarity matrix.

- ϵ ball graph: join vertices within a radius ϵ
- k-NN graph: join the k-NN vertices.
- fully connected graph: join all the vertices; S can be something like $S_{il} = \exp(-\frac{||x_i - x_l||^2}{2\sigma^2})$ then we establish a threshold σ to decide if two points are connected or not. $d_{ij} \leq \sigma \rightarrow$ not connected



Naive approach: the k-way cut

Define C as a set of vertices, $C \subset V$, and $C \neq \emptyset$. The cut is defined as:

$cut(C, \bar{C}) = \sum_{i \in C} \sum_{l \in \bar{C}} S_{il}$

$cut(C_1, C_2, ..., C_k) = \frac{1}{2} \sum_m cut(C_m, \bar{C_m})$

this is the formula we are interested in minimizing for a given cluster partition.

Applied like this, this method would result in a lot of single points labeled as clusters and one big cluster in the middle.

Balancing the k-way cut

- take into account the size of the clusters → minimize the **ratio cut**
  - $|C_m|$: number of vertices in cluster C_m

$$\text{ratio\_cut}(C_1, C_2, ..., C_k) = \frac{1}{2} \sum_m \frac{cut(C_m, \bar{C_m})}{|C_m|}$$

- take into account the size of the clusters and the similarity between the points (both inter and intra cluster) → it tries to minimize a sort of **normalized cut**. Group these two concepts into the volume of the cluster, obtained through the degree $g_i = \sum_l S_{il}$
  - $Vol(C_m) = \sum_{i \in C_m} g_i$
  - $normalized\_cut(C_1, C_2, ..., C_k) = \frac{1}{2} \sum_m \frac{cut(C_m, \bar{C_m})}{Vol(C_m)}$

The path to spectral clustering

- S our symmetric **matrix of weights**, of size $N \times N$.

$$S = \begin{bmatrix} 0 & S_{12} & ... & S_{1N} \\ S_{21} & 0 & ... & S_{2N} \\ ... & ... & ... & ... \\ S_{N1} & S_{N2} & ... & 0 \end{bmatrix}$$

- D the diagonal **matrix of degrees**, of size $N \times N$. $D_{ii} = g_i$, $D_{ij} = 0$ if $i \neq j$.

$$D = \begin{bmatrix} g_1 & 0 & ... & 0 \\ 0 & g_2 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & g_N \end{bmatrix}$$

  The degree of a node is how many edges connect to it (weighted by distance): $g_i = \sum_l S_{il}$.

The Laplacian is just another matrix representation of a graph. It has several beautiful properties, which we will take advantage of for spectral clustering. To calculate the normal Laplacian (there are several variants), we just subtract the adjacency matrix from our degree matrix.

- $L = D - S$ the **Laplacian matrix** of size $N \times N$.
  - L is symmetric as well
  - it is positive semi-definite, the lowest eigenvalue is 0 and the others are ≥ 0 (one element in matrix D is symply given by a linear combination of the other elements)

The Laplacian's diagonal is the degree of our nodes, and the off diagonal is the negative edge weights. This is the representation we are after for performing spectral clustering.

Eigenvalues of Graph Laplacian

As mentioned, the Laplacian has some beautiful properties. To get a sense for this, we can see that when the graph is completely disconnected, all our eigenvalues are 0. As we add edges, some of our eigenvalues increase. In fact, the number of 0 eigenvalues corresponds to the number of connected components in our graph.

The first eigenvalue is always 0 because we have at least one connected component (the whole graph is connected).

The first nonzero eigenvalue is called the spectral gap. The spectral gap gives us some notion of the density of the graph.

If we take a vector v with dimension N, we obtain the following:

$$v^T L v = v^T D v - v^T S v = \sum_i g_i \cdot v_i^2 - \sum_{il} v_i v_l S_{il}$$

$$= \frac{1}{2}(2\sum_i v_i^2 g_i - 2\sum_{il} v_i v_l S_{il})$$

$$= \frac{1}{2}(\sum_i v_i^2 g_i - 2\sum_{il} v_i v_l S_{il} + \sum_l v_l^2 g_l)$$

$$= \frac{1}{2}\sum_{il} S_{il}(v_i - v_l)^2$$

where the last step is determined by the fact that $g_i = \sum_l S_{il}$.

This result will be useful later.

How to minimize the loss (the cost of the cut)

Let's see how to minimize the loss we defined before, in the case of ratio cut with k = 2.

Minimize

$$\text{ratio\_cut} = \frac{cut(C, \bar{C})}{|C|} = \frac{\sum_{i \in C} \sum_{l \in \bar{C}} S_{il}}{|C|}$$

Let's define an indicator vector f:

- if $i \in C \rightarrow f_i = \sqrt{\frac{|\bar{C}|}{|C|}}$
- if $i \in \bar{C} \rightarrow f_i = -\sqrt{\frac{|C|}{|\bar{C}|}}$

we have that $f^T L f = \frac{1}{2} \sum_{il} S_{il}(f_i - f_l)^2$

- if $i \in C$ and $l \in C \rightarrow f_i - f_l = 0$
- if $i \in \bar{C}$ and $l \in \bar{C} \rightarrow f_i - f_l = 0$
- if $i \in C$ and $l \in \bar{C} \rightarrow f_i - f_l = \sqrt{\frac{|\bar{C}|}{|C|}} - (-\sqrt{\frac{|C|}{|\bar{C}|}})$

$$f^T L f = \frac{1}{2} \sum_{il} S_{il}(f_i - f_l)^2 = N \text{ratio\_cut}(C, \bar{C})$$

So minimizing the ratio cut, is equivalent to minimizing $f^T L f$ and, to minimize it, we have to find the eigenvector corresponding to the second smallest eigenvalue of the Laplacian matrix.

Taking it would mean to cut the graph in two parts, in the best way possible, that is choosing the minimum graph cut needed to separate the a part of the graph (remember that if we already have two connected components, the second eigenvalue would be zero and then choosing the smallest no zero eigenvalue means to further divide the components we have).

If we want k clusters, we will take k eigenvectors, to find the best cut between them. We will end up with k *connected components*, which are our clusters.

Each element in a eigenvector corresponds to a node in the graph, and the sign of the element tells us to which cluster the node belongs, wrt the corresponding cut.

The size of the eigenvector is, indeed, n (the number of nodes in the graph).

Finally, by performing **k-means** on the coordinates of the eigenvectors, it is pretty straightforward to obtain the clusters corresponding to the sum of the cuts.

We are given two constraints for the indicator vector f:

- $||f|| = \sqrt{N}$
- $f^T 1 = 0$, f perpendicular to the unit (1 vector of ones)

These constraints allow us to avoid the trivial solution in which all the elements belong to the same cluster, that is to say choosing always the smallest eigenvalue, which is zero.

The ratio cut for arbitrary k

- k indicator vectors $\rightarrow$ a matrix H of size $N \times k$.
- $H_{il} = \dfrac{1}{\sqrt{|C_l|}}$ if $i \in C_l$, $H_{il} = 0$ otherwise, where $i \in \{1, 2, ..., N\}$ and $l \in \{1, 2, ..., k\}$
- we also impose the identity $H^T H = I$ (the columns are orthonormal)
- the indicator vector for the cluster l is $H_{(\cdot,l)} = h_l$ (the whole column)
- Then we have that: $h_l^T L h_l = \dfrac{cut(C_l, \bar{C_l})}{|C_l|} = (H^T L H)_{ll}$

so the ratio cut:

$$\text{ratio\_cut}(C_1, ..., C_k) = trace(H^T L H) = \sum_l \frac{cut(C_l, \bar{C_l})}{|C_l|}$$

Problems

In the way we defined the problem the indicator vector has discrete values. But minimizing a discrete space is extremely difficult, so we want to remove this constraint.

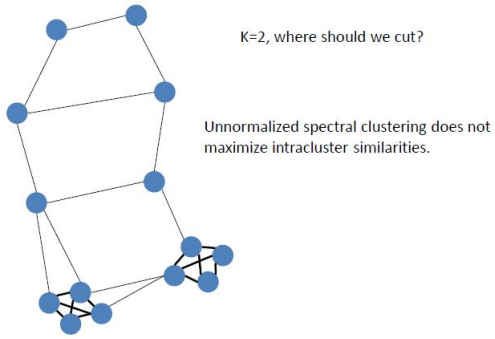
- Indicator matrix will have real values and the problem will be solved by the eigenvectors of the Laplacian matrix.
- Once the minimization is solved, we will have to discretize the solution (using k-means on the eigenvectors coordinates).
- There is no warranty that the solution to this is similar to the unrelaxed one.

Unnormalized spectral clustering

1. construct similarity graph;
2. compute the Laplacian matrix;
3. compute the first k eigenvectors of the Laplacian matrix;

4. use the coordinates as input for the k-means algorithm and obtain clusters.

Unnormalized spectral clustering does not take into accunt intra-cluster similarity.



Normalized spectral clustering

In order to address the problem of the unnormalized spectral clustering, we can use the normalized cut.

$$normalized\_cut(C, \bar{C}) = \frac{cut(C, \bar{C})}{Vol(C)} + \frac{cut(C, \bar{C})}{Vol(\bar{C})}$$

Analogously to the unnormalized case, we can define the indicator matrix H and the indicator vector h_l.

$H_{il} = \frac{1}{Vol(C_l)}$ if $i \in C_l$, $H_{il} = 0$ otherwise

where $i \in \{1, 2, ..., N\}$ and $l \in \{1, 2, ..., k\}$

In order to minimize the normalized cut, we have to minimize the trace $Tr(H^T LH)$, given that $H^T DH = I$. In this case the condition we are imposing to indicator matrix is different ($H^T DH = I$) and we won't have anymore an eigenvalue-eigenvectors problem.

How to deal with this?

We take $T = D^{1/2}H$ so that minimizing the trace $Tr(H^T LH)$ is equal to minimize the trace $Tr(T^T D^{-1/2} L D^{-1/2} T)$, with $T^T T = I$.

We modifying the Laplacian, taking $L_{symm} = D^{-1/2} L D^{-1/2}$, the following steps are the same as before.

Final considerations:

- the spectral clustering is closely related to kernel k-means.

- with a different transformation, the Laplacian can be transformed in a random walk matrix and related with a Markov state model analysis.
- the value of k can be inferred from a gap in te spectrum of eigenvalues.

How to generate the similarity matrix? One of main concerns about spectral clustering is how to build the initial graph.

Spectral clustering problems

- How to construct the similarity graph?
- How to decide the number of clusters in absence of a gap?
- There are also problems with computational complexity (difficult to scale).
- Speactral clustering is sensitive to noise and outliers.

Modern Clustering Algorithms

Lecture 23/5/23

We are going to see two other ways to do clustering.

Affinity Propagation

- Affinity propagation creates clusters by sending **messages** between pairs of samples until convergence.

  Two kind of messages are exchanged: **responsibility** and **availability**.

  Responsibility: accounts for the accumulated evidence that point k should be the exemplar for point i. ($\rightarrow$)

  $$r(i,k) \leftarrow S_{ik} - \max_{k' \neq k}\{a(i,k') + S_{ik'}\}$$

  Availability: accounts for the accumulated evidence that point i should choose point k as its exemplar. ($\leftarrow$)

  $$a(i,k) \leftarrow \min\{0, r(k,k) + \sum_{i' \notin \{i,k\}} \max\{0, r(i',k)\}\}$$

  $$a_{k,k} \leftarrow \sum_{i' \neq k} \max\{0, r(i',k)\}$$

  Note that it considers the values for all the other samples.

- It aims to determine the samples that can be exemplars for clusters

The exchange of responsibility and availability messages goes on iteratively.

- S_{ik} is the **similarity** between point i and point k, usually it is taken $S_{i,k} = -d_{ik}$, where d_{ik} is the distance between point i and point k (e.g. $||x_i - x_k||^2$).
- S_{kk}, known as **preference**, plays an important role in the algorithm. A value close to the minimum possible similarity produces fewer classes, while

a value close to the maximum possible similarity produces many clusters. It is typically set to the median similarity value of the input similarities.

Algorithm

1. The algorithm starts by setting all the values of r and a to zero.
2. Iterate by first updating the responsibility matrix r and then the availability matrix a, until a certain number of iterations is reached.
   - after the first iteration:

$$r(i,k) \leftarrow S_{i,k} - \max_{k' \neq k}\{S_{ik'}\}$$

$$a(i,k) \leftarrow \min\{0, r(k,k) + \sum_{i' \notin \{i,k\}} \max\{0, r(i',k)\}\}$$

   And at time t:

$$r(i,k)^t \leftarrow S_{i,k} - \max_{k' \neq k}\{a(i,k')^{t-1} + S_{i,k'}\}$$

   The responsibility decreases when the availability for other points increases.

- The exemplars, in the end, are extracted from the final matrices as those whose 'responsibility + availability' is positive $r(k,k) + a(k,k) > 0$.
- The element i is assigned to the exemplar k if $r(i,k)$ is the maximum among all the $r(i,k)$.

Unstable behavior

- The algorithm described in this way is usually **not stable numerically**.
- A **dumping factor** is introduced to smooth the transitions.

$$\begin{cases} r(i,k)_{t+1} = \lambda r(i,k)_t + (1-\lambda) r(i,k)_{t+1} \\ a(i,k)_{t+1} = \lambda a(i,k)_t + (1-\lambda) a(i,k)_{t+1} \end{cases}$$

The dumping factor descreases the speed with which we update our parameters, so that if the change was not correct, the error is not propagated too much and will be hopefully corrected in the next iteration.

Prices and caveats

- Two parameters to tune: **dumping factor** and **preference**, and number of iterations.
- The algorithm is sensitive to the choice of preference and how to set it is not so clear.

- The method is computationally expensive, with a complexity of $O(N^2T)$, where N is the number of samples and T is the number of iterations until convergence.
- It tends to generate convex cluster.

It was shown this method works really well on some specific (Economic) datasets, with applying also some specific knowledge of the field and the problem to solve.

Second method for clustering we'll see today is Expectation Maximization for clustering.

Expectation Maximization clustering

(Expectation Maximization is also a way to estimate parameters in density estimation.)

- Consider your data as a set of **realizations** of an underlying probability function $p(x)$.
- Assume the functional form (e.g. Gaussians) of $p(x)$ and estimate its parameters by maximum likelihood.

So, for example we will start with some parameters α_0, β_0 and iteratively update them to maximize the likelihood of the data.

Until we find $\alpha_{opt}, \beta_{opt}$, when reaching convergence.

Then, when we have the optimal parameters (in the case of two sets of parameters (α, β) we assumed that the data were generated by two different Gaussian), we can assign each point to the Gaussian that (with highest probability) generated it.

In this way, we have a sort of clustering algorithm.

The kind of output that we obtain from EM is a kind of *fuzzy* output, where for each point we have a probability of belonging to each cluster.

- EM is an iterative procedure to compute the **Maximum Likelihood** estimate, even in presence of missing or hidden data.
- **EM** constists of two steps:
  - **E-step**: estimate the expected population corresponding to each (basis) function given the current model parameters.
  - **M-step**: compute the model parameters that maximize the likelihood of the data given the expected population.

We will see one simple example of EM clustering, the **Gaussian Mixture Model**.

What is a mixture of K Gaussians?

$$p(x) = \sum_{k=1}^{K} \pi_k F(x|\mu_k, \Sigma_k)$$

with

$$\sum_{k=1}^{K} \pi_k = 1$$

where π_k is the mixing coefficient, μ_k is the mean and Σ_k is the covariance matrix of the k-th Gaussian.

μ_k, Σ_k will be θ_k

$F(x|\theta_k)$ is the Gaussian distribution with parameters θ_k.

If all points $x \in X$ are obtained by mixtures of K Gaussians then:

$$p(X) = L(\pi, \theta) = \prod_{i=1}^{n} p(x^i) = \prod_{i=1}^{n} \sum_{k=1}^{K} \pi_k F(x^i|\theta_k)$$

and the goal is finding $\pi_1, ..., \pi_k$ and $\theta_1, ..., \theta_k$ such that the likelihood $p(X)$ of the data is maximized.

Log likelihood as well:

$$\mathcal{L}(\pi, \theta) = \sum_{i=1}^{n} ln\Big(\sum_{k=1}^{K} \pi_k F(x^i|\theta_k)\Big)$$

How does this work?

- Every point x^i is **probabilistically** assigned (generated) to the k-th Gaussian.
- Probability that point x^i is generated by the k-th Gaussian is:

$$w_{ik} = \frac{\pi_k F(x^i|\theta_k)}{\sum_{j=1}^{K} \pi_j F(x^i|\theta_j)}$$

Once we have this, we can start exploiting these weights to estimate the parameters of the Gaussians.

- Every Gaussian (cluster) has an effective (weighted) number of points assigned to it, which is the sum of the weights of the points assigned to it:

$$N_k = \sum_{i=1}^{n} w_{ik}$$

- and has mean:

$$\mu_k = \frac{1}{N_k} \sum_{i=1}^{n} w_{ik} x^i$$

- and variance:

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{n} w_{ik} (x^i - \mu_k)(x^i - \mu_k)^T$$

So we have a sort of population (provided by weights), through which we can estimate the parameters of the Gaussians.

Then we can update the weights and iterate until convergence.

Algorithm

- Initialize the means, variances (θ_k) and the mixing coefficients π_k and evaluate the initial value of the log likelihood.
- **Expectation step**: evaluate the weights w_{ik} for each point x^i and each Gaussian k.

$$w_{ik} = \frac{\pi_k F(x^i|\theta_k)}{\sum_{j=1}^{K} \pi_j F(x^i|\theta_j)}$$

- **Maximization step**: update the parameters of the Gaussians using the weights w_{ik}.

$$\mu_k^{new} = \frac{1}{N_k} \sum_{i=1}^{n} w_{ik} x^i$$

$$\Sigma_k = \frac{1}{N_k} \sum_{i=1}^{n} w_{ik} (x^i - \mu_k^{new})(x^i - \mu_k^{new})^T$$

  And:

$$\pi_k^{new} = \frac{N_k}{N}$$

- Then re-evaluate $\mathcal{L}_{\backslash]\sqsupseteq}(\pi, \theta)$ and stop if we converged.

What it is usually done for the initialization is to use the K-means algorithm to initialize the means (and variances) of the Gaussians.

EM characteristics

- Notice the similarity between EM for Normal mixtures and K-means: the expectation step is the assignment and the maximization step is the update of the centers.
- If the model is not realistic, the results won't be meaningful.
- There is no guarantee that the algorithm will converge to the global maximum of the likelihood.
- K can be inferred with bayesian model selection → **Dirichlet Process Mixture Model**.

Lecture 24/5/23

Recap on clustering

- **k-means clustering**: hard partitioning, convex clusters, fast, sensitive to initialization, sensitive to outliers.
- **c-means clustering**: fuzzy partitioning
- **hierachical clustering**: we obtain a dendogram, which can be seen as subset of clusters
  - single linkage
  - complete linkage
  - average linkage
  - centroid linkage
  - Ward's method
  - DIANA (divisive analysis)

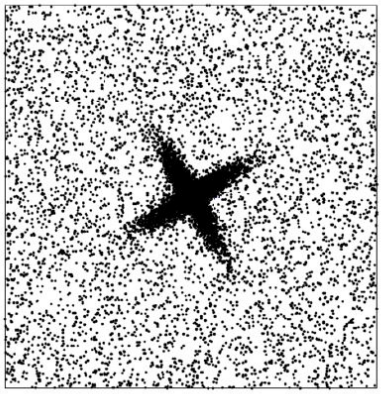
These above are **classical clustering methods**, then we have also seen:

- **Kernel k-means**
- **Spectral Clustering**
- **Affinity Propagation**
- **EM Clustering**

Problems of EM

EM is a **parametric model**, that is to say it imposes a model on the data.

eg. of a situation where EM fails:



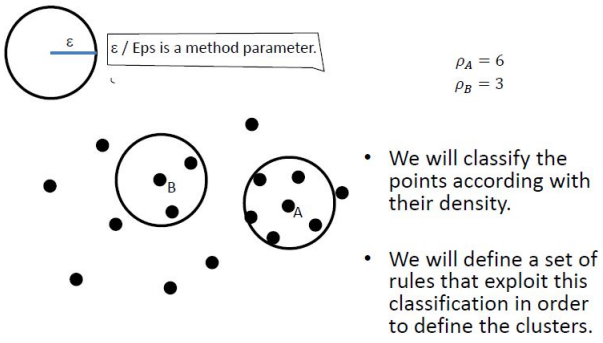
Here EM would capture the one unique cluster as two different ones.

Today we are going to see **non-parametric density based clustering methods**.

Density Based Spatial Clustering of Applications with Noise (DBSCAN)

Density-based clustering locates regions of high density that are separated from one another by regions of low density.

density: number of points within a specified radius ϵ.



- A point is a **core point** if it has more than a specified number of points (MinPts) within ϵ. These are points that are the interior of a cluster.
- A **border point** has fewer than MinPts within ϵ but is in the neighborhood of a core point. These are points that are on the edge of a cluster.
- All other points are **noise points**.

New parameters: ϵ and MinPts.

Other definitions:

- ϵ-neighborhood of a point p: $N_\epsilon(p) = \{q \in D | dist(p,q) \leq \epsilon\}$, objects within a radius of ϵ from p.
- **core objects**: are those whose ϵ-neighborhood contains at least MinPts objects.

What DBSCAN does with these definitions?

- Any two core points that are close enough– within a distance Eps of one another – are put in the same cluster
- Any border point that is close enough to a core point is put in the same cluster as the core point
- Noise points are discarded

Reachability

- Directly density-reachable: an object q is directly-reachable from object p if q is within the ϵ-neighborhood of p and p is a **core object**.
- Density-reachable: an object p is density-reachable from q wrt ϵ and MinPts if there is a chain of objects $p_1, .., p_n$, with $p_1 = q$, $p_n = p$ such that p_{i+1} is directly density-reachable from p_i wrt ϵ and MinPts for $i = 1, .., n-1$.

  This essentially means that there is a path of core points from q to p.
- Density-connected: an object p is density-connected to q wrt ϵ and MinPts if there is an object o such that both p and q are density-reachable from o wrt ϵ and MinPts.

All the points that density connected should be put in the same cluster.

Concepts: cluster and noise

- **Cluster**: a cluster C, in a set of objects D wrt ϵ and MinPts, is a non-empty subset of D satisfying:
  - **Maximality**: for every objects p, q, with $p \in C$, and q density-reachable from p wrt ϵ and MinPts, then $q \in C$.
  - **Connectivity**: for all $p, q \in C$, p is density-connected to q wrt ϵ and MinPts in D
  - Cluster contains core objects as well as border objects.
- **Noise**: objects that are not in any cluster (which are not directly density-reachable from at least one core object) are considered noise.

Algorithm

- select a point p
- retrieve all points density-reachable from p wrt ϵ and MinPts

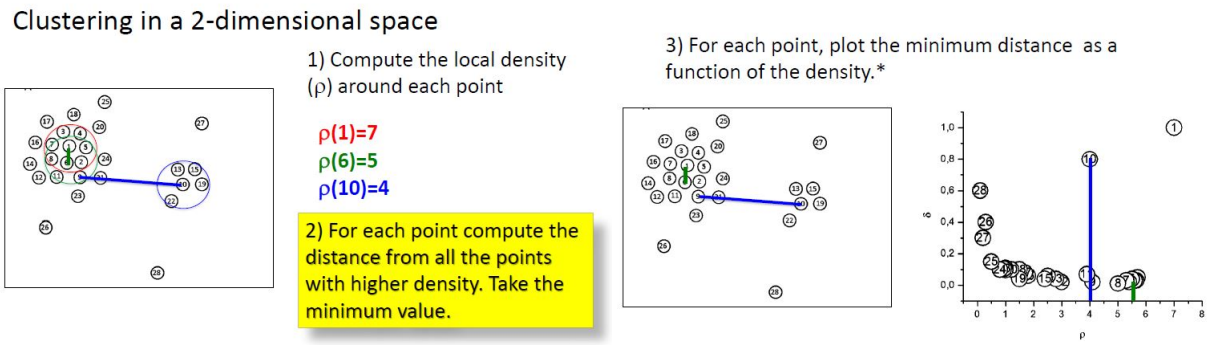
- if p is a core point, a cluster is formed
- if p is a border point, no points are density-reachable from p, then visit the next point of the database
- continue the process until all of the points have been processed

Result is independent of the order of the points in the database.

Fast Search and Find of Density Peaks

Clusters: peaks in the density of data points (peaks in the "mother" probability distribution)

Density Peaks decision graph



The point with higher density has the largest delta value by convention.

4. The outliers in this graph are the cluster centers.
5. Assign each point to the same cluster of its nearest neighbor of higher density (in increasing order of density).

Density Peaks Algorithm

1. given a distance matrix d_{ij}, for each data point i compute the associated density:

$$\rho_i = \sum_j \chi(d_{ij} - d_c) \sim \sum_j e^{-(d_{ij}/d_c)^2}$$

   where χ is the characteristic function, d_c is a distance threshold.

2. Sort data points in order of decreasing density and compute the minimum distance from a point with higher density.

$$\delta_i = \min_{j:\rho_j > \rho_i} d_{ij}$$

3. The delta for the first element can be arbitrarily assigned at the end as 5% higher than the maximum delta.
4. Generate the decision graph and identify the outliers.
5. Assign each point to the same cluster of its nearest neighbor of higher density.

There are some data points that are not assigned to any cluster, these are the outliers. Those associated to a density lower than the saddle point are tagged as 'halo'.

Border definition

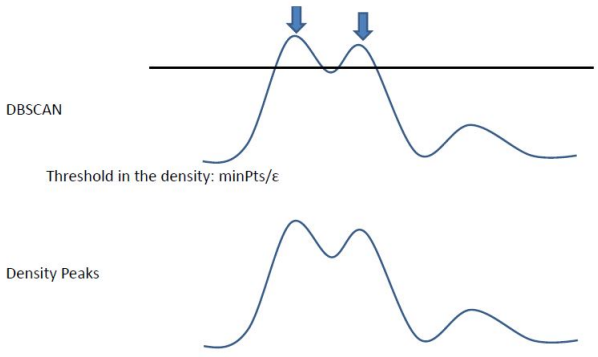
- A point i of cluster A is border point if it has, within d_c, a point j belonging to another cluster.
- The border density of A is, then, the higher density among all the border points belonging to A.

Last two steps of the algorithm:

6. Check the neighborhoods of each data points and assign as **border points** those that have an element of other cluster within d_c.
7. Find the border density of a cluster j as the higher density among the border points belonging to this cluster. All the points that belong to cluster j and have density lower than the border density are tagged as halo.

Differences between DBSCAN and Density Peaks clustering

- in DBSCAN we put a threshold (minPts/ϵ) to identify the regions of high density, while in Density Peaks we have more **flexibility** because we use the density of the points to identify the regions of high density.



You can see in the figure above, that for DBSCAN there is no way of setting a threshold that allow us to identify all the clusters.

- However, in Density Peaks we have to choose the threshold d_c to identify the regions of high density, which means that we have to *know* which clusters we are looking for.

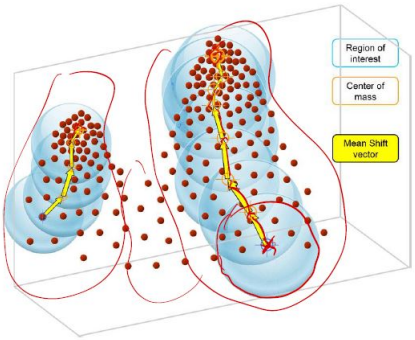
Mean Shift clustering

- The idea is similar to Density-Peaks: Find the density maxima and assign the data points to its correspondent maxima.
- The density is estimated with **kernel density estimation**.
- For each point, shifts its coordinates towards the weighted mean of the density in a region of interest until it converges.

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

Where N is the region of interest, K is the kernel function.

- If the kernel is flat (Parzen windows), $m(x)$ is the average position of the points within the region of interest.



Problems

- It depends again on one hyperparameter: the size and the number of the region of interest.
- The densities obtained this way can oscillate a lot, which means that the convergence is not guaranteed.
- This algorithm works with data defined in specific coordinate space, we cannot just work with distances as in previous algorithms.

Cluster validation

Supervised classification:

- Class labels known for ground truth
- Accuracy, precision, recall

Cluster analysis:

- No class labels

Validation need to:

- Compare clustering algorithms
- Solve number of clusters
- Avoid finding patterns in noise

We can do two kinds of validation:

- **External validation**: (we have an external classification, but we don't have explicit labels)
  - if we have it, validate against ground truth
  - otherwise compare directly the clusters (measuring how similar they are)
- **Internal validation**: try to know how our data are distributed
  - validate without external information
  - it works with different number of clusters
  - solve the number of clusters

Cluster validation process

1. Distinguishing whether non-random structure actually exists in the data (one cluster).
2. Comparing the results of a cluster analysis to externally known results, e.g., to externally given class labels.
3. Evaluating how well the results of a cluster analysis fit the data without reference to external information.
4. Comparing the results of two different sets of cluster analyses to determine which is better.
5. Determining the number of clusters.

Internal indexes

Ground truth is rarely available but unsupervised validation must be done.

Trying to minimizes (or maximizes) internal index:

- Variances of within cluster and between clusters
- Rate-distortion method
- F-ratio
- Davies-Bouldin index (DBI)
- Bayesian Information Criterion (BIC)
- Silhouette Coefficient
- Minimum description principle (MDL)
- Stochastic complexity (SC)

One index we have already seen is the **Mean Squared Error** (MSE), which is nothing else than the loss function of the K-means algorithm.

However there are some problems with this index, in particular wrt the representation of hierarchical or not convex clusters.

From MSE to cluster validity

Once we have our MSE index, we can play with it trying to: minimize within cluster variance and maximize between cluster variance.

- **Within cluster variance**:

$$SSW(C,k) = \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - \mu_k||^2$$

- **Between cluster variance**:

$$SSB(C,k) = \sum_{k=1}^{K} n_k ||\mu_k - \mu||^2$$

  where μ is the mean of all the data points and n_k is the number of points in cluster k.

Then, the total variance of the data set is:

$$\sigma(X) = SS(C,k) = SSW(C,k) + SSB(C,k)$$

Through these indexes, we can define the following way of validating a clustering algorithm.

F-ratio variance test

Variance-ratio F-test measures ratio between-groups variance against the within-groups variance (original f-test).

In our case:

$$F = \frac{k \sum_{k=1}^{K} \sum_{x_i \in C_k} ||x_i - \mu_k||^2}{\sum_{k=1}^{K} n_k ||\mu_k - \mu||^2} = \frac{k \cdot SSW}{SSB}$$

where N is the number of data points.

This is also called **WB-index** and it monotonously decreases when the number of clusters increases.

The main problem is that we need to work with coordinates and it is also why another index was proposed: **silhouette coefficient**.

Silhouette coefficient, Kaufman and Rousseeuw (1990)

It depends on two concepts:

- **Cohesion**: how closely related are objects in a cluster

  $a(x)$, average distance of x to all other points in the same cluster.

- **Separation**: how distinct or well-separated a cluster is from other clusters

  $b(x)$ average distance of x to the vectors in other clusters. We take the minimum of the average distance to all other clusters.

The silhouette coefficient is defined as:

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

The silhouette coefficient is a number between -1 and 1. The higher the value, the better the clustering.

Silhouette has a preference for convex clusters, however we can use it to compare different clustering algorithms.

External indexes

If true class labels (ground truth) are known, the validity of a clustering can be verified by comparing the class labels and clustering labels.

Then we can define the following indexes:

- **Pair counting**: it measures the number of pairs of objects that are grouped or separated in the same way wrt different classifications.

  Measure the number of pairs that are in:

  Same class **both** in *P* and *G*.

  $$a = \frac{1}{2}\sum_{i=1}^{K}\sum_{j=1}^{K'} n_{ij}(n_{ij} - 1)$$ ✓

  Same class in *P* but different in *G*.

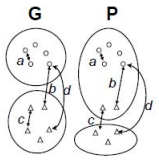
  $$b = \frac{1}{2}(\sum_{j=1}^{K'} n_{\cdot j}^2 - \sum_{i=1}^{K}\sum_{j=1}^{K'} n_{ij}^2)$$ ✗

  Different classes in *P* but same in *G*.

  $$c = \frac{1}{2}(\sum_{i=1}^{K} n_{i\cdot}^2 - \sum_{i=1}^{K}\sum_{j=1}^{K'} n_{ij}^2)$$ ✗

  Different classes **both** in *P* and *G*.

  $$d = \frac{1}{2}(N^2 + \sum_{i=1}^{K}\sum_{j=1}^{K'} n_{ij}^2 - (\sum_{i=1}^{K} n_{i\cdot}^2 + \sum_{j=1}^{K'} n_{\cdot j}^2))$$ ✓

  Based on this it was defined the **Rand index**:

$$R(P, G) = \frac{a + d}{a + b + c + d}$$

  where:

agreement: a, d

disagreement: b, c

The problem of Rand index is that it is not normalized, so it does not go to zero just due to random noise.

So a way to solve this problem is to use the **Adjusted Rand index**, which accounts for what would happen with a random classification in order to normalize the index.

$$ARI = \frac{RI - E[RI]}{1 - E[RI]}$$

where $E[RI]$ is the expected value of the Rand index.

- **Information theoretic**: it measures the **mutual information** between two classifications.

  Normalized Mutual Information (NMI) is a normalized version of the mutual information (MI) score, which is a measure of the overlap between two clusters.

$$MI(k, G) = \sum_{l=1}^{K} \sum_{j=1}^{G} p(l, j) log \frac{p(l, j)}{p(l)p(j)}$$

where $p(k, g)$ is the joint probability of a point being in cluster k and in class g, $p(k)$ is the probability of a point being in cluster k and $p(g)$ is the probability of a point being in class g.

The problem with MI is that it is not bounded, for this reason we use NMI.

$$NMI(k, G) = \frac{2MI(k, G)}{H(k) + H(G)}$$

where $H(k)$ and $H(G)$ are the entropies of the two classifications.

$$H(k) = -\sum_{k=1}^{K} p(k) log[p(k)]$$