Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Clustering by Affinity Propagation

Sebastian F. Walter[1]

Department of Physics, ETH Zürich

A thesis submitted for the degree of

Diploma in Physics (ETH)

Supervised by

Dr. Bernd Fischer, Prof. Dr. Joachim M. Buhmann

May - August 2007

[1]email: sebastian.walter@gmail.com

Acknowledgements

This work wouldn't have been possible without the great support from the whole Machine Learning group.

I am especially grateful to Bernd Fischer for his patience and guidance. He not only devoted much of his time to discuss difficulties of my work, but also turned out to be an enthusiastic and committed discussion partner. He also helped me to find the right direction when I had lost view of the big picture.

My thanks also go to Professor Joachim Buhmann who even sacrified two of his lunch breaks to discuss problems that arose during my work.

Furthermore, I'd like to thank Thomas Fuchs and Steven Armstrong for their help with various technical problems. I'd also like to thank Theodor Mader for this moral support and discussions that helped to view the problem from different angles. Finally, I'd like to thank Barbara Keller for proof-reading my chapter on molecular biology.

Contents

List of Notation

Notation	Definition	p.
x^*	the optimal configuration of an objective function; e.g., $x^* = \text{argmin}_x f(x)$	10
H	objective function; H stands for Hamiltonian	11
M	binary matrix; $M_{kn} := \delta_{kc_n}$	11
D	dimension	11
d	index corresponding to the d'th dimensions. $d \in \{1, \ldots, D\}$	11
N	number of data points	11
n	index corresponding to the n'th datapoint. $n \in \{1, \ldots, N\}$	11
K	number of clusters	11
k	index corresponding to the k'th cluster. $k \in \{1, \ldots, K\}$	11
δ	Kronecker delta function: $\delta_{kl} = 1$ if $k = l$ and zero otherwise	11
x_n	the n'th data point. $x_n \in \mathbb{R}^D$	11
y_k	centroid of cluster k	11
s	$s(y_k, x_n)$ is the similarity between x_n and y_k. If $s(y_1, x_n) > s(y_2, x_n)$ then y_1 is more similar to x_n than y_2	11
d	$d_{nm} = -s_{nm}$ is the dissimilarity between two data points n and m	11
c	cluster label, assigns each data point n to a cluster k, $c = (c_1, \ldots, c_N)$, $c_n \in \{1, \ldots, K\}$	11
$\lvert C_k \rvert$	number of data points in the k'th cluster C_k; $\lvert C_k \rvert := \sum_{n=1}^N M_{kn}$	11
$\bar{x}$	estimated expectation value of a random variable; e.g., $\bar{x}_k := \frac{1}{\lvert C_k \rvert} \sum_{n=1}^N M_{kn} x_n$ is the average of data points in cluster k	12
pdf	probability density function	14
D^{KL}	Kullback-Leibler divergence; $D^{\text{KL}}(h_n \lvert\lvert q_k) := \sum_{d=1}^D h_{nd} \log \frac{h_{nd}}{q_{kd}}$	14
e	exemplar label; $e_n \in \{1, \ldots, N\}$ is the data point e_n that data point n picks as exemplar	15
T	(computational) temperature; it is a Lagrange parameter	18
Z	partition function, $Z = \sum_c H(c)$ the sum over all possible configurations	18
F	the free energy; $F = U - TS$	27
MPF	marginalize a product-function	31
AP	affinity propagation	31
MPF	marginalize a product-function	31
OSF	optimize a sum-function	31
$\mathbb{Y}$	a set, for example the set of real numbers $\mathbb{R}$	31
$\oplus$	binary operator; for example the $\min(\cdot, \cdot)$ operator. usually $\oplus$ is commutative, i.e. $x \oplus y = y \oplus x$	31
$\odot$	binary operator, for example the addition operator $+$. possibly non-commutative	31
$\bigodot$	generalized product, i.e., $\bigodot a_i = a_1 \odot a_2 \odot \cdots \odot a_J$	31
$\bigoplus$	generalized sum, i.e., $\bigoplus a_i = a_1 \oplus a_2 \oplus \cdots \oplus a_J$	31
$[\cdot]$	Iverson's notation,[true] = 1 and [false] = 0	34
$\mathbb{1}$	indicator function, $\mathbb{1}$ returns 1 if the expression in the subscript is true and returns 0 otherwise	41
a	a_{nm} is the weight of an edge from n to m of an adjacency matrix A	50
$\text{cut}(C_k, V \setminus C_k)$	sum of edge weights a_{nm} that need to be cut to separate the vertices n in cluster C_k from the rest of the graph; $\text{cut}(C_k, V \setminus C_k) = \sum_{n \in C_k} \sum_{m \in V \setminus C_k} a_{nm}$	50
$\text{assoc}(C_k, V)$	association is the sum of all outgoing edges from vertices in C_k, i.e., $\text{assoc}(C_k, V) = \sum_{n \in C_k} \sum_{m \in V} a_{nm}$	50
$\tilde{M}$	weighted binary matrix; $\tilde{M}_{kn} = w_n M_{kn}$, where w_n is a weight	51
$\lvert \tilde{C} \rvert$	weighted size of a cluster; $\lvert \tilde{C}_k \rvert = \sum_{n=1}^N \tilde{M}_{kn}$	51
K	kernel matrix; $K = XX^T$	55
PBC	path-based clustering	57

1 Introduction

In almost all scientific fields (computer science, physics,...) one is regularly confronted with the problem to find the configuration that optimizes a function. In this thesis, we compare different approaches to optimize objective functions as they arise in the clustering problem. We have a particular interest in the performance of a recently published clustering algorithm called affinity propagation (AP) [FD07a]. It has been reported that AP yields much better results than other algorithms when applied to certain problems; and that in only a fraction of the runtime. We investigate to what extend the reported results are reproducible and in what cases the application of this algorithm yields better results than competing methods.

For the K-means problem, we find that AP performs at least as well as the competing algorithms in terms of quality. However, due to a memory footprint of $\mathcal{O}(N^2)$, the algorithm cannot be applied on datasets where the number of data points N is large. Another reason why AP is not very suited for large N is its $\mathcal{O}(N^2)$ scaling of the runtime per iteration. The K-means algorithm and deterministic annealing (DA) have a runtime that scales with $\mathcal{O}(NKD)$. Therefore, when the dimension D and number of clusters K is small, DA and K-means have a much lower runtime. We observe, that AP's runtime is mostly independent of the dimension D and the number of clusters K. That means, when K and D is large, e.g $K = 50$ and $D = 100$, AP can be much faster than K-means algorithm and DA. Also, the K-means algorithm is not only slow for large K but has severe problems to find good solutions. Hence, AP works well in settings where the K-means algorithm has problems. Compared to hierarchical clustering algorithms, e.g. Ward's method, AP generally runs much slower. When clusters are well-defined and there is only little noise in the dataset, the performance is comparable. If that is not the case, AP finds better solutions.

We test how well AP works on non-symmetric similarities between data points. An example, where such non-symmetric similarities arise, is histogram clustering with the negative Kullback-Leibler divergence as similarity measure. For this ACM objective function, the solution found by AP is only little worse than the hard assignment ACM algorithm. However, AP can work only on rather small images of $\approx 60 \times 60$ due to the memory limits. The runtime is also considerably higher.

More elaborate methods for image segmentation are graph clustering methods. One of most popular methods is the normalized cut criterion (ncut). One can reformulate the ncut problem to a weighted K-means problem. The weighted K-means problem can be solved with AP. Unfortunately, AP didn't work well for ncut with a radial basis functions (RBF) kernel; it also didn't work well for kernel K-means with an RBF kernel.

We also test how well AP works for path-based clustering. We observe that it works reasonably well; but not as well as the K-means algorithm on the embedded dataset. It has the advantage, that no embedding in a high-dimensional space has to be computed, i.e., one can save the PCA.

Though AP is not the perfect choice for all problems, there are cases when it clearly outperforms other methods; especially when there are many clusters. We therefore try to apply the same framework and tricks, that have been used to derive AP, to existing objective functions. We investigate if it is possible to derive an AP-style algorithm for pairwise clustering. Unfortunately, the structure of the pairwise clustering objective function makes it impossible to derive an efficient AP-style algorithm. Therefore, we attempt to approach the problem differently by searching for objective functions that would allow an efficient AP-style implementation. This is not an easy task: One has to find a useful objective function under the constraint that the structure allows an efficient implementation in principle and under the constraint that one can analytically optimize the message passing.

The thesis is structured as follows: In Section 2 we place the clustering problem in the context of optimization resp. marginalization of objective functions. We explain the goals and steps that have to be taken for a complete clustering analysis. In Section 3 we define some objective functions with focus on the K-means objective function and some additional information on how some of these objective functions relate to each other. In Section 4 we

introduce the theory behind two standard clustering algorithms: In Section 4.1 the K-means algorithm, in Section 4.2 Ward's method. In Section 5we explain simulated and deterministic annealing, in Section 6 hard and soft assignment ACM. Finally, in Section 7 we introduce affinity propagation. After that, we explain the objectives of cluster validation and show how affinity propagations internal cluster validation compares to a method as CLEST (Section 8). Then we introduce in Section 9 the framework how the pairwise clustering objective function can be mapped to a K-means problem in a high-dimensional space. In Section 10 we show how graph clustering, at the example of ncut, can be mapped to a weighted pairwise clustering objective function. The weighted pairwise clustering function is equivalent to a weighted K-means problem which can be solved with AP. Since the results were not very nice, we test how well AP works on kernel K-means (Section 11). AP has apparent problems to solve this optimization problem. In Section 12 we make a comparison between AP and the K-means algorithm when the data points lie on a $2D$ hyperplane in a 100 dimensional vector space and the 98 remaining dimension are normal distributed noise with increasing variance. We find that AP soon fails to give reliable clustering solutions whereas the K-means algorithm still can. After that, we show how AP can be applied to solve the path-based clustering problem in Section 13. In Section 14 we investigate if the framework and tricks used for the derivation of AP can be applied to other objective functions. In Section 15 we explain our design goals of our implementation and why we have chosen Python and C++ as programming languages. We briefly explain how Python can be extended with C++ code, which proved to be very nice in practice. Additionally we show a simple example how C++-code can be inlined in Python code to circumvent certain performance bottlenecks. Finally, we show experimental results obtained for color image segmentation (Section 17), clustering of microarray data (Section 18) and for the K-means problem on toy datasets (Section 19).

2 The Clustering Problem

2.1 Posing The Clustering Problem

In unsupervised learning, we are posed the problem to structure data into groups such that all members of a group are somehow similar. To make a simplistic example: Neanderthal man sees trees, grass, sabertooth and bear. Though he might not have had a teacher who told him "this is an animal and this is a plant", he can tell that sabertooth and bear are more similar to each other than to tree and grass. His measure of similarity could possibly be: sabertooth and bear are both fluffy, or rather: they have big, sharp teeth while grass and plant haven't. He has to make a *model* of what he sees.

Now, Neanderthal man is not particularly bright; his wife Neanderthal woman is much cleverer. Even with the same measure of similarity, Neanderthal man cannot interpret what he sees as good as good as his wife. I.e., his wife finds a solution closer to the *optimum*. For example, let's assume that Neanderthal man cannot keep two things in his mind at the same time: either he can compare animals by their color, or he can compare them by their teeth. Thus, he might partition bunny rabbit and bear into one group and sabertooth to another; simply because sabertooth is buff while bunny and bear are brown.

There is also the question, how many groups make sense. While plants and animals might be considered two groups, one can always find differences between members of a group and therefore one could place each species in its own group. The question is: How many groups make sense? This is the question of the *validity* of a clustering solution.

To wrap it up: The *clustering problem* is actually three problems:

1. The *modeling problem* is the task to find an appropriate model that maps the physical reality to a mathematical formulation. This problem is ill-posed since it is subjective. Typically, modeling includes a *choice of a space*, *selection of features*, *standardization of data* and the choice of an *objective function*.

2. The *optimization problem* is the task to optimize the objective function. The objective functions are usually highly non-convex functions. Therefore, the task of finding the globally optimal solution is generally a hard problem to solve. The actual attempt to solve the problem is done by an algorithm. An *algorithm* is a deterministic sequence of instructions that, given an initial state, finds a well-defined solution in a finite number of steps. Since the problem is so hard, the algorithms usually do not find (and often do not even attempt) to find the globally best solution. One is therefore happy if they are heuristics that give good solutions to the optimization problem.

3. In the *model selection problem* it is the goal to find the best set of parameters that have previously been fixed in the optimization. A typical example is the number of clusters K. As the modeling problem, the model selection problem is ill-posed because the judgment of *good* and *bad* is subjective. Once again, one resorts to map the model selection problem to a mathematical model.

2.2 The Modeling Problem

The modeling of clustering problems faces the same fundamental problem as science in general: We do not have direct access to the physical reality. Information on reality can only be inferred from observations resp. measurements. On the way, information is lost. To make things worse, the underlying process is possibly non-deterministic. Additionally, other processes and interactions may perturb the measurements.

In the clustering problem one assumes that there are groups, i.e., each data point has a label that uniquely defines to which group it belongs. However, the cluster labels are not observable. It is the goal to retrieve the cluster label again. This is often possible, because there are features that differ from group to group. The task of the modeling is to find appropriate features and define rules that allow an inference from the features to the cluster labels. There are two major ideas what those rules may be. In the *parametric modeling* one assumes that there is a special underlying process that seperates the groups in the feature space. The task is to estimate the parameters that explain the observations. For example, it is often assumed that some process produced Gaussian shaped clusters; but the mean and covariance are unknown and have to be inferred. An example for *non-parametric* clustering is superparamagnetic clustering [BWD96]. One does not assume a special underlying generative process. Instead, a grouping is attempted based solely on the structure of the data. Explicitly, superparamagnetic clustering works by estimating pair-correlations between data points at varying temperatures.

Fig. 2.1: The fundamental problem of science is the inference on the physical reality from observable (measurable) quantities. The modeling problem is the task to make measurements that allow reliable inference.

2.3 The Optimization Problem

The *optimization problem* can be stated as follows: Find

$$x^* = \text{argmax}_x f(x) \,, \tag{2.1}$$

where x is a finite or continuous variable. The function $f(x)$ is called the *objective function*. x^* denotes the optimal configuration. An exampe is the *maximum a posteriori* (MAP) estimation of parameters θ such that $\theta^*_{\text{MAP}} = \text{argmax}_\theta p(x|\theta)p(\theta)$. If the priors $p(\theta)$ are unknown, they are assumed to be uniformly distributed. Then the problem is equivalent to a *maximum likelihood* estimation $\theta^*_{\text{ML}} = \text{argmax}_\theta p(x|\theta)$. The positions are defined as $x = (x_1, \ldots, x_N)$, $x_n \in \mathbb{R}^D$. The parameters are $\theta = (\theta_1, \ldots, \theta_K)$. For mixture of Gaussians a typical parameter is $\theta_k = (\mu_k, \text{cov}_k)$, where μ_k is the mean and cov is the covariance of the Gaussian.

3 Objective Functions

In this section we introduce various objective functions, which we call Hamiltonian H. One can distinguish two types of clustering objective functions H: Those that work on vectorial data (e.g. the K-means objective function) and those who work on pairwise distances (e.g. normalized cut).

3.1 The K-Means Objective Function

The importance of the K-means problem comes from five facts: 1) compared to other cost functions it is mathematically relatively simple to treat, 2) it is the limiting case of many different objective functions (e.g. it can be viewed as pairwise clustering problem as well as zero temperature mixture of Gaussians with iid variances in each dimension). 3) The clustering solution is subjectively good because it finds compact clusters and is robust to outliers. 4) Furthermore, there exist fast heuristics that give good results in low running time; the heuristics are usually also rather easy to implement. (5) Finally, is it often possible to map other clustering problems to a (weighted) K-means problem.

The *K-means objective function* H^{KM} can be written in different ways that are equivalent.

The hybrid formulation Find K *centroids* $y = (y_1, \ldots, y_K)$ and N *cluster labels* $c = (c_1, \ldots, c_n)$ for each of the N data points $x_n \in \mathbb{R}^D$ s.t. the Hamiltonian $H(y, c)$ is minimized:

$$H^{\mathrm{KM}}(y, c) := -\sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn} s(y_k, x_n) \;, \tag{3.1}$$

where $M_{kn} := \delta_{k,c_n}$ is a binary matrix. N denotes the number of data points and K denotes the number of clusters. We use the notation, that variables defining the range of sums are printed in upper case and their corresponding iterators in lower case. I.e., the data points range from $n = [1, \ldots, N]$. δ is the Kronecker delta function. The function $s(y_k, x_n)$ is called the *similarity* between the *centroid* y_k and the *data point* x_n. It is defined s.t. if $s(y_1, x_n) > s(y_2, x_n)$ then y_1 is more similar to x_n than y_2. Note that there exist situations when $s(x_n, y_k) \neq s(y_k, x_n)$; for example when the similarity is the negative Kullback-Leibler divergence. For the K-means problem the similarity is symmetric and is defined as

$$s(y_k, x_n) := -\|y_k - x_n\|_2^2 \tag{3.2}$$

where $\|\cdot\|_2$ is the Euclidian norm. The important point of (Eqn. 3.1) is that c is uniquely defined by y and vice versa. This leads naturally to iterative algorithms as for example the K-means algorithm and also deterministic annealing.

The discrete formulation considers the centroids y_k to be uniquely defined by the cluster labels c_n. The objective function is then given by

$$\begin{aligned} H^{\mathrm{KM}}(c) &:= -\sum_{n=1}^{N}\sum_{k=1}^{K} M_{kn} s(y_k(c), x_n) \\ &= -\sum_{n=1}^{N}\sum_{k=1}^{K} M_{kn} \|\frac{1}{|C_k|}\sum_{m=1}^{N} M_{km} x_m - x_n\|_2^2 \;, \end{aligned} \tag{3.3}$$

where $|C_k| = \sum_{n=1}^{N} M_{kn}$ is the number of datapoints in cluster k . If s is the negative squared Euclidian distance then y can be analytically calculated from c and the problem is of purely combinatorial nature.

The continuous formulation of the K-means problem is given as

$$\begin{aligned} H^{\mathrm{KM}}(y) &= -\sum_{n=1}^{N}\sum_{k=1}^{K} M_{nk}(y)s(y_k, x_n) \\ &= -\sum_{n=1}^{N} \min_k s(y_k, x_n) \ , \end{aligned} \tag{3.4}$$

and is therefore also known as *min-sum* clustering.

3.1.1 Relation Between Discrete and Continuous Formulation

Here we show how the discrete and continuous formulation relate.

discrete → continuous If the cluster labels c are known, then the centroids y_k can be calculated analytically as the average over all data points in cluster k. The hybrid K-means objective function states that we have to find y such that H is minimized assuming c is known. This can be seen from the following derivation [DFK+99]:

$$\begin{aligned} \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|x_n - y_k\|_2^2 &= \sum_{k=1}^{K}\sum_{n=1}^{N} \|x_n - \bar{x}_k + \bar{x}_k - y_k\|_2^2 \\ &= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\left(\|x_n - \bar{x}_k\|_2^2 + \|\bar{x}_k - y_k\|_2^2 - 2\langle x_n - \bar{x}_k | \bar{x}_k - y_k\rangle\right) \\ &= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|x_n - \bar{x}_k\|_2^2 + |C_k|\|\bar{x}_k - y_k\|_2^2 - 2\sum_{n=1}^{N} M_{kn}\langle x_n - \bar{x}_k | \bar{x}_k - y_k\rangle \\ &= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|x_n - \bar{x}_k\|_2^2 + |C_k|\|\bar{x}_k - y_k\|_2^2 - 2\langle \underbrace{\sum_{n=1}^{N} M_{kn}x_n - |C_k|\bar{x}_k}_{=0} | \bar{x}_k - y_k\rangle \\ &= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|x_n - \bar{x}_k\|_2^2 + |C_k|\|\bar{x}_k - y_k\|_2^2 \ , \end{aligned} \tag{3.5}$$

where $|C_k| := \sum_{n=1}^{N} M_{kn}$ is the number of data points in cluster k. In words: The intra cluster sum of distances $\sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|y_k - x_n\|_2^2$ is minimal for $y_k = \bar{x}_k := \frac{1}{|C_k|}\sum_{n=1}^{N} M_{kn}x_n$.

continuous → discrete On the other hand, if y_k are known, then the configuration c is also uniquely defined:

$$\begin{aligned} c^* &= \mathrm{argmin}_c \sum_{n=1}^{N} \|y_{c_n} - x_n\|_2^2 \\ &= \sum_{n=1}^{N} \mathrm{argmin}_{c_n} \|y_{c_n} - x_n\|_2^2 \ , \end{aligned}$$

i.e., a data point is assigned to the nearest centroid y_k.

3.2 The K-Medoids Objective Function

When the centroids y_k are restricted to be chosen from data points x_n, one calls the K-means problem the *K-medoids* problem. The objective function is given by

$$H^{\mathrm{K-medoids}}(e) := -\sum_{n=1}^{N}\sum_{k=1}^{K} M_{mn}s(x_m, x_n) \ , \tag{3.6}$$

where $M_{mn} = \delta_{e_m n}$ and $e_m \in [1, \ldots, N]$ is the *exemplar label*.

3.3 Pairwise Clustering Objective Function

The *pairwise clustering* problem has as objective function

$$H^{\mathrm{PW}}(c) \;=\; -\frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N} s(x_m,x_n)\frac{M_{kn}M_{km}}{|C_k|}\;, \tag{3.7}$$

where the binary matrix $M_{kn} = \delta_{kc_n}$ holds the information about the configuration of the system.

The pairwise clustering function is equivalent to the K-means objective function if the similarities $s(x_m, x_n)$ come from a positive definite kernel matrix as explain in Section 9. If $-s_{nm} = \|x_n - x_m\|_2^2$ this condition is naturally fulfilled. We start similarly to [Fis06]:

$$\begin{aligned}
H^{\mathrm{KM}}(c) &= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\|y_x - x_n\|_2^2 \\
&= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\left(\langle y_k|y_k\rangle + \langle x_n|x_n\rangle - 2\langle y_k|x_n\rangle\right) \\
&= \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\sum_{m=1}^{N}\sum_{o=1}^{N}\frac{1}{|C_k|^2}M_{ko}M_{km}\langle x_m|x_o\rangle + \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\langle x_n|x_n\rangle - 2\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{km}M \\
&= -\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{km}M_{kn}\langle x_m|x_n\rangle + \sum_{k=1}^{K}\sum_{n=1}^{N} M_{kn}\langle x_n|x_n\rangle \\
&\overset{(\&)}{=} \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{kn}\langle x_n|x_n\rangle - \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{km}M_{kn}\langle x_m|x_n\rangle \\
&= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{km}\langle x_n - x_m|x_n - x_m + x_m\rangle \\
&= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{km}\langle x_n - x_m|x_n - x_m\rangle + \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{km}\langle x_n - x_m|x_m\rangle \\
&\overset{(\dagger)}{=} \underbrace{\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{km}\langle x_n - x_m|x_n - x_m\rangle}_{=2H^{\mathrm{PWC}}} - \underbrace{\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{1}{|C_k|}M_{kn}M_{km}\langle x_n - x_m|x_n\rangle}_{H^{\mathrm{Kmeans}}}\;,
\end{aligned}$$

In line (†) with line (&) we see the equation $H^{\mathrm{KM}} = 2H^{\mathrm{PWC}} - H^{\mathrm{KM}}$ that we solve for H^{KM}. We obtain as result the equivalence of the K-Means and the pairwise clustering objective function:

$$H^{\mathrm{KM}} = -\frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{M_{kn}M_{km}}{|C_k|}\|x_m - x_n\|_2^2 = H^{\mathrm{PW}} \tag{3.9}$$

3.4 The Kullback-Leibler ACM Objective Function

Sometimes, the data points are histograms (resp. probability density functions (pdf)) and not vectors in a vector space. A quite natural distance between histograms (reps. pdfs) is the Kullback-Leibler divergence. The *Kullback-Leibler (KL) divergence* D^{KL} is defined as

$$D^{\mathrm{KL}}(h_n||q_k) := \sum_{d=1}^{D} h_{nd}\log\frac{h_{nd}}{q_{kd}}\;, \tag{3.10}$$

where h_{nd} corresponds to the histogram associated with data point n. D stands for the dimension, i.e. the number of bins in a histogram. It is a measure of dissimilarity between two

histograms h_d and q_d. Since it is not symmetric, it is not a distance. However, it has the nice property that $D^{\mathrm{KL}}(h||q) \geq 0$. The limit $D^{\mathrm{KL}}(h||q) = 0$ is assumed iff the histograms q_d and p_d are the same.

The objective function that we try to optimize is the sum of all KL divergences from the centroids q_k to the datapoints h_n, i.e.

$$H(c) = \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} D^{\mathrm{KL}}(h_n||q_k) \; . \tag{3.11}$$

With the above definition of the objective function (and not with $D^{\mathrm{KL}}(q_k||h_n)$ instead), the optimization problem can be simplified:

$$\begin{aligned}
c^* &= \mathrm{argmin}_c \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} D^{\mathrm{KL}}(h_n||q_k) \\
&= \mathrm{argmin}_c \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log \frac{h_{nd}}{q_{kd}} \\
&= \mathrm{argmin}_c \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} (\log h_{nd} - \log q_{kd}) \\
&= \mathrm{argmin}_c \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \underbrace{\sum_{d=1}^{D} h_{nd} \log h_{nd}}_{=:Z_n} - \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd}) \\
&= \mathrm{argmin}_c \sum_{n=1}^{N} Z_n \underbrace{\sum_{k=1}^{K} M_{kn}}_{=1} - \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd}) \\
&= \mathrm{argmin}_c - \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd} \; .
\end{aligned}$$

Therefore, the *ACM objective function* is defined as

$$H^{\mathrm{ACM}}(c) = - \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd} \tag{3.12}$$

In the following, when we refer to the ACM problem, we mean the optimization of the above objective function. In the literature, ACM stands for *asymmetric clustering methods*.

3.5 Affinity Propagation's Objective Function

Frey proposed the following objective function

$$H(e) = - \sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{m=1}^{N} \delta_m(e) \; , \tag{3.13}$$

$$\delta_m(e) = \begin{cases} \infty & \text{if } e_m \neq m \text{ and there exists a } l \text{ s.t. } e_l = m \\ 0 & \text{else} \end{cases} \tag{3.14}$$

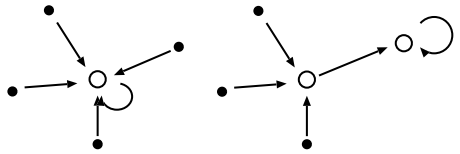
$$e^* = \mathrm{argmin}_{e \in \mathbf{e}} H(e) \; , \tag{3.15}$$

where $e_n \in [1, \ldots, N]$ is the *exemplar label* which assigns each data point n to another data point e_n. This data point is called an *exemplar*. In Figure 3.1, valid and invalid configurations are depicted. An exemplar has to pick itself as exemplar. The self-similarity of a data point is called the *preference* since it indicates how much the data point would like to be an exemplar. If

no prior information about the preferences is known, the preferences should be set to a common value. In Section 7 we investigate how the preferences influence the number of identified clusters and show the relation between this cost function and the K-means cost function. If AP has found the number of clusters K it has also tried to minimize the K-means objective function as following calculation shows:

$$\begin{aligned}
\min_e H^{\mathrm{AP}}(e) &= \min_e \left[-\sum_{n=1}^{N} s(e_n, n) + \sum_{m=1}^{N} \delta_m(e) \right] \\
&= \min_e \left[-\left(\sum_{n=1}^{N} \tilde{s}(e_n, n) + \sum_{m=1}^{N} \delta_m(e) \right) - K(e)p \right] \\
&= \min_K \left[\min_e H^{\mathrm{K-Medoids}}(e) - Kp \right] \qquad (3.16)
\end{aligned}$$

where $\tilde{s}$ means, that the self-similarities are set to zero, i.e., the self-similarities p are pulled out. We therefore obtain the additional factor Kp. This corresponds to minimal description length prior.



(a) (b)

Fig. 3.1: In a) a valid configuration is depicted since the exemplar picks itself as exemplar. b) shows an invalid configuration because the chosen exemplar has picked another data point as his exemplar.

4 Standard Algorithms for the K-Means Problem

In this section, we introduce several algorithms and their theoretical foundation. The algorithms are later used as comparison to affinity propagation.

4.1 The K-Means Algorithm

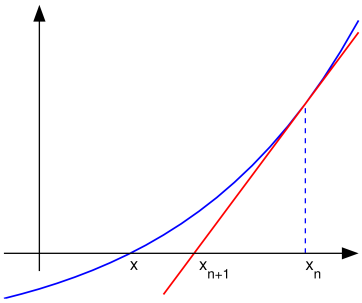
The K-Means algorithm belongs to the class of greedy iterative methods that try to solve the optimization problem $x^* = \mathrm{argmin}_x f(x)$, where f is the function to be optimized and x a discrete or continuous variable. The problem can often be approached by picking an initial guess $x^{(0)}$ as starting condition and iterate $x^{(t+1)} = \mathrm{argmin}_x \tilde{f}[x^{(s)}](x)$, where $\tilde{f}[x^{(s)}]$ is a simplified function based on a known solution $x^{(s)}$ such that in the next timestep $(t+1)$ $x^{(t+1)}$ is closer to the optimum. s may be a previous or future timestep which makes the problem explicit or implicit at each step. A typical example is the Newton method (Figure 4.1).

Applied to the K-means problem [Eqn. (3.1)] this works in the following way:

- In an *E-Step*: estimate the cluster labels c_n with y fixed, i.e., $c^{(t-1)} = \mathrm{argmin}_c H(c, y = y^{(t-1)})$.
- In an *M-Step*: solve simplified function based on the solution obtained in the previous timestep, i.e. $y^{(t)} = \mathrm{argmin}_y H(c = c^{(t-1)}), y)$.

Taking the minimum in the M-step can be calculated analytically and y_k is simply the average of all datapoints x_n in cluster k. The final algorithm is summarized in Algorithm 1.

The runtime of the K-means algorithm is $\mathcal{O}(NKD)$ per iteration. Usually, the K-means algorithm converges very fast (of order ≈ 10).



(a) Newton method.

Fig. 4.1: The Newton method simplifies the function at each step by approximating it with a linear curve.

while *not converged* **do**
  foreach $n \in \{1, \ldots, N\}$ **do**
    $c_n = \text{argmin}_k \|y_k - x_n\|^2$
  foreach $k \in \{1, \ldots, K\}$ **do**
    $y_k = \frac{1}{|C_k|} \sum_{n=1}^{N} M_{kn} x_n$

Algorithm 1: The K-means algorithm.

4.2 Hierarchical Agglomerative Clustering

In biology and chemistry, data has often a structure that allows representation as a tree. For example in the set {grass, tree, eagle, cat, dog}, {grass, tree} can be put in the group "plants" while {eagle,cat,dog} belong to the group "animals". The group {eagle,cat,dog} can be furthed grouped in birds {eagle} and mammals {cat,dog}. There are two approaches that find such a grouping in a tree: divisive and agglomerative. We decided to use *hierarchical agglomerative clustering* (HAC) algorithms. A HAC algorithm works by recursively combining exactly two clusters at each step. *Ward's method* picks those two clusters that will lead to the smallest increase in the K-means objective function whereas the *average linkage* algorithm joins the two clusters with the smallest average sum of pairwise inter cluster distances. A short calculation below shows the differences between the criterion. We included the average linkage algorithm to show that the clustering solution strongly depends on the objective function that is optimized.

Initialization:
$c_k = k \; \forall k \in \{1, \ldots, N\}$
Iteration:
for $K' = N$ *downto* K **do**
  $(k^*, l^*) = \text{argmin}_{k,l} \Big\{ \sum_{n \in C_k \cup C_l} \| \frac{|C_l|}{|C_l|+|C_k|} y_l + \frac{|C_k|}{|C_l|+|C_k|} y_k - x_n \|_2^2$
  $- \sum_{n \in C_k} \|y_k - x_n\|_2^2 - \sum_{n \in C_k} \|y_k - x_n\|_2^2 \Big\}$
  $y_k = \frac{|C_{l^*}|}{|C_{l^*}|+|C_{k^*}|} y_{l^*} + \frac{|C_{k^*}|}{|C_{l^*}|+|C_{k^*}|} y_{k^*}$
  foreach $n \in C_{l^*}$ **do**
    $cl_n = k^*$
  $y_{l^*} = y_K$
  foreach $n \in C_{K'}$ **do**
    $c_n = l^*$

Algorithm 2: Ward's method. At each step two clusters are combined s.t. the resulting K-means objective function H^{KM} is as small as possible. This algorithm is for example implemented in R's library `stats`. R has an interface to Python (called RPy).

Initialization:
$c_k = k \; \forall k \in [1, N]$
Iteration:
for $K' = N$ *downto* K **do**
  $(k^*, l^*) = \text{argmin}_{k,l} - \frac{1}{|C_l||C_k|} \sum_{n \in C_l} \sum_{m \in C_k} s(x_n, x_m)$
  foreach $n \in C_{l^*}$ **do**
    $cl_n = k^*$
  foreach $n \in C_{K'}$ **do**
    $c_n = l^*$

Algorithm 3: The average linkage algorithm combines two clusters with the smallest average inter cluster sum of distances between data points. It does not require vectorial data in an Euclidean vector space, i.e., it can work directly on a similarity matrix resp. distance matrix. It is implemented for example in the C-cluster library [Thec].

4.2.1 The Relation Between the Average Linkage Algorithm and Ward's Method

The calculation shows the difference between the average linkage algorithm and Ward's method:

$$\begin{aligned}
& H^{\text{K-Means}}(C_1 \cup C_2) - H^{\text{K-Means}}(C_1) - H^{\text{K-Means}}(C_2) \\
= \; & \frac{1}{|C_1| + |C_2|} \sum_{n \in C_1 \cup C_2} \sum_{m \in C_1 \cup C_2} d_{nm} - \frac{1}{|C_1|} \sum_{n \in C_1} \sum_{m \in C_1} d_{nm} - \frac{1}{|C_2|} \sum_{n \in C_2} \sum_{m \in C_2} d_{nm} \\
= \; & \frac{1}{|C_1| + |C_2|} \left[\sum_{n \in C_1} \sum_{m \in C_1} d_{nm} + \sum_{n \in C_2} \sum_{m \in C_2} d_{nm} + \sum_{n \in C_1} \sum_{m \in C_2} d_{nm} + \sum_{n \in C_2} \sum_{m \in C_1} d_{nm} \right] \\
& - \frac{1}{|C_1|} \sum_{n \in C_1} \sum_{m \in C_1} d_{nm} - \frac{1}{|C_2|} \sum_{n \in C_2} \sum_{m \in C_2} d_{nm} \\
\overset{(\dagger)}{=} \; & \frac{|C_2|}{|C_1| + |C_2|} \sum_{n \in C_1} \sum_{m \in C_1} d_{nm} + \frac{|C_1|}{|C_1| + |C_2|} \sum_{n \in C_2} \sum_{m \in C_2} d_{nm} + \frac{2}{|C_1| + |C_2|} \sum_{n \in C_1} \sum_{m \in C_2} d_{nm} \qquad (4.1)
\end{aligned}$$

where we have used the notation $H(C)$ for the intra cluster sum of cluster C. d_{nm} is the squared Euclidean distance. In (†) one can see that Ward's method cannot be the same as average linkage where the combined clusters are found by $(k^*, l^*) = \text{argmin}_{k,l} - \frac{1}{|C_l||C_k|} \sum_{n \in C_l} \sum_{m \in C_k} s(x_n, x_m)$.

4.2.2 Derivation of Ward's Method

Similarly to the calculation in the previous section:

$$\begin{aligned}
H^{\text{K-Means}}(C_1 \cup C_2) &= \| \underbrace{\frac{1}{|C_1| + |C_2|} \sum_{m \in C_1 \cup C_2} x_m}_{=y} - x_n \|_2^2 \\
&= \| \frac{1}{|C_1| + |C_2|} \left[\sum_{m \in C_1} x_m + \sum_{m \in C_2} x_m \right] - x_n \|_2^2 \\
&= \| \frac{1}{|C_1| + |C_2|} \left[\frac{|C_1|}{|C_1|} \sum_{m \in C_1} x_m + \frac{|C_2|}{|C_2|} \sum_{m \in C_2} x_m \right] - x_n \|_2^2 \\
&= \| \frac{1}{|C_1| + |C_2|} \{ |C_1| y_1 + |C_2| y_2 \} - x_n \|_2^2 \\
&= \| \frac{|C_1|}{|C_1| + |C_2|} y_1 + \frac{|C_2|}{|C_1| + |C_2|} y_2 - x_n \|_2^2 \; . \qquad (4.2)
\end{aligned}$$

5 Annealing Methods

In this section we introduce the theory for a class of algorithms that map the optimization problem to the problem of estimating certain expectation values and marginal probabilities that depend on a computational temperature T. At the limit of $T = 0$ the estimation of probabilities is equivalent to the optimization problem. The algorithms start with a high temperature and gradually lower it to $T = 0$ in the hope to avoid local minima.

5.1 Simulated Annealing

Consider that we want to solve the following optimization problem:

$$c^* = \text{argmin}_c H(c) \; . \tag{5.1}$$

Simulated annealing (SA) is a generic optimization technique that aims for the global optimum. The basic idea is to treat the configuration c as random variable with a known probability mass $p_T(c)$ that depends on a Lagrange parameter T. T stands for the (computational) temperature. At $T = 0$ this probability mass must have the property that the optimal coniguration c^* has probability $p(c^*) = 1$. One probability distribution that can be used is the Boltzmann distribution (aka Gibbs distribution)

$$p_T^{Gibbs}(c) = \frac{1}{Z} \exp(-\frac{1}{T} H(c)) \; , \tag{5.2}$$

where the normalization constant $Z = \sum_c \exp(-\frac{1}{T} H(c))$ is called the *partition function*. We have therefore mapped the problem of finding the configuration c^* that optimizes $H(c)$ to the problem of finding configuration where the joint probability $p_T(c)$ at $T = 0$ is one.

In practice, the exact inference is impossible and one has either to resort to deterministic approximation schemes or one can use a *Monte Carlo* method. The idea is to take samples from all possible configurations and use them to estimate expectation values of random variables. For example in solid state physics, one wishes to know the magnetization of a model system for a direct comparison to a real world experiment. The magnetization is defined as expectation value of a spin, i.e. $m = \mathbb{E}[s_i]$. However, the computation of the partition function Z is usually in NP. One therefore replaces static Monte Carlo methods by dynamic versions.

This is done by a *Markov Chain Monte Carlo* method: New sample configurations c^{new} are not drawn from the probability distribution p but according to a transition probability that is conditioned by the previously sampled configuration c^{old}. This transition probability must fulfill two requirements:

1. It must allow to reach every configuration in finite steps, i.e. it must be *ergodic*.
2. In the limit of infinite updates, the probability of each configuration must be $p(c)$. A sufficient condition is the *detailed balance* condition $p(c^{\text{old}})p(c^{\text{new}}|c^{\text{old}}) = p(c^{\text{old}}|c^{\text{new}})p(c^{\text{new}})$.

The *Metropolis algorithm* proposes the transition probability

$$p(c^{c^{\text{new}}|\text{old}}) = \begin{cases} 1 & \text{if } \Delta H < 0 \\ \exp(-\frac{1}{T}\Delta H) & \text{else} \end{cases} \; , \tag{5.3}$$

where $\Delta H := H(c^{\text{new}}) - H(c^{\text{old}})$ and T is the temperature of the system. The Metropolis algorithm is ergodic and fulfills the detailed balance condition. The expected probability of a configuration is given by the Boltzmann distribution. At low temperature has a problem though: The autocorrelation time is very long. That means, if T is small and one is trapped in a local minimum, one has to wait very, very long until the system leaves the local minimum again. The idea is to start at a high temperature that allows the system to easily change the configuration, even if that configuration corresponds to a local minimum of the objective function H. There is a theoretical proof that states that if the cooling schedule is logorithmic (i.e. $T^{n+1} = T^0 / \log n$) then the annealing process halts almost certainly at the global optimum. Unfortunately this result is of little use in practice, because the runtime of such a cooling schedule is too long and one could as well traverse the whole search space.

```
Initialize state c randomly
while T > T_min do
    while not converged do
        propose new state c^new from a proposal distribution Q
        accept transition with a probability that respects the detailed balance condition
    lower temperature T according to a cooling schedule
```

Algorithm 4: The simulated annealing (SA) algorithm (general version). The proposal distribution Q is for example the single site update: pick a random cluster label c_n and randomly assign it to cluster k. More elaborate proposal distributions as implemented in the Swendsen-Wang or Wolff algorithm can help to overcome critical slowing downs at continuous phase transitions.

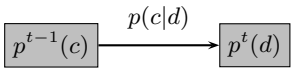


Fig. 5.1: A Markov chain. The basic idea is to generate the probability distribution $p(c)$ as limiting case of a Markov chain, i.e., $p(c) = \lim_{t\to\infty} p^t(c)$, where $p^t(c) = \sum_d p(c|d)p^{t-1}(d)$. For convergence a necessary condition is $p(c) = \sum_d p(c|d)p(d)$. The detailed balance condition is a sufficient condition for the transition probability $p(c|d)$ to converge to $p(c)$. In the pictorial representation of the Markov chain, c and d stand for all possible configurations.

5.2 Deterministic Annealing

In the literature, there are two possible approaches to *deterministic annealing* (DA); one suggested by Rose [Ros98] and one suggested by Hofmann [HB97]. Here, we choose the approach by Hofmann and Buhmann since its derivation is shorter, more generic and has closer ties to SA.

5.2.1 Derivation of the Deterministic Annealing Algorithm

The probability of a configuration is given by $p(c) = \frac{\exp\left(-\beta \sum_{n=1}^{N} \|y_{c_n} - x_n\|_2^2\right)}{\sum_c \exp\left(-\beta \sum_{n=1}^{N} \|y_{c_n} - x_n\|_2^2\right)}$, i.e., the probability mass function $p(c)$ depends uniquely on the centroids y. At each temperature, one has therefore to find the centroids y^* that minimize the free energy, i.e.,

$$y^* = \operatorname{argmin}_y F(y) \; . \tag{5.4}$$

This is an implicit equation that suggests the use of an expectation maximization ansatz.

- In an *E-Step* estimate the new probability distribution with fixed centroids y_k:

$$\begin{aligned} p(c) &= \frac{\exp\left(-\beta H(c)\right)}{\sum_c \exp\left(-\beta H(c)\right)} \\ &= \frac{\exp\left(-\beta \sum_{n=1}^N \|y_{c_n} - x_n\|_2^2\right)}{\sum_c \exp\left(-\beta \sum_{n=1}^N \|y_{c_n} - x_n\|_2^2\right)} \\ &= \frac{\exp\left(-\beta \sum_{n=1}^N \|y_{c_n} - x_n\|_2^2\right)}{\sum_c \prod_{n=1}^N \exp\left(-\beta \|y_{c_n} - x_n\|_2^2\right)} \\ &\overset{(*)}{=} \frac{\exp\left(-\beta \sum_{n=1}^N \|y_{c_n} - x_n\|_2^2\right)}{\prod_{n=1}^N \sum_{c_n} \exp\left(-\beta \|y_{c_n} - x_n\|_2^2\right)} \\ &= \prod_{n=1}^N \frac{\exp\left(-\beta \sum_{n=1}^N \|y_{c_n} - x_n\|_2^2\right)}{\sum_{c_n} \exp\left(-\beta \|y_{c_n} - x_n\|_2^2\right)} \\ &= \prod_{n=1}^N p(c_n|n) \;, \end{aligned} \tag{5.5}$$

  where the step performed in $(*)$ is a tremendous computational simplification of the problem because the sum over all possible configurations can be omitted. Then, it is only necessary to estimate

$$p(k|n) = \frac{\exp\left(-\beta \|y_k - x_n\|_2^2\right)}{\sum_{l=1}^K \exp\left(-\beta \|y_l - x_n\|_2^2\right)} \;. \tag{5.6}$$

  for each data point n.

- In an *M-Step* minimize the free energy under fixed configuration p:

$$\begin{aligned} y^* &= \operatorname{argmin}_y F = \operatorname{argmin}_y - T \ln Z \\ &= \operatorname{argmax}_y \ln \prod_{n=1}^N \sum_{d_n} \exp\left(-\beta \|y_{d_n} - x_n\|_2^2\right) \\ &= \operatorname{argmax}_y \sum_{n=1}^N \ln \sum_{d_n} \exp\left(-\beta \|y_{d_n} - x_n\|_2^2\right) \end{aligned} \tag{5.7}$$

  To find the y that minimizes F we take the derivative

$$\begin{aligned} 0 &\overset{!}{=} \sum_{n=1}^N \nabla_{y_k} \ln \sum_{d_n} \exp\left(-\beta \|y_l - x_n\|_2^2\right) \\ &= \sum_{n=1}^N \frac{-2\beta (y_k - x_n) \exp\left(-\beta \|y_k - x_n\|_2^2\right)}{\sum_{d_n} \exp\left(-\beta \|y_{d_n} - x_n\|_2^2\right)} \\ &= \frac{-2\beta \left[y_k \sum_{n=1}^N \exp\left(-\beta \|y_k - x_n\|_2^2\right) - \sum_{n=1}^N x_n \exp\left(-\beta \|y_k - x_n\|_2^2\right)\right]}{\sum_{d_n} \exp\left(-\beta \|y_{d_n} - x_n\|_2^2\right)} \;, \end{aligned} \tag{5.8}$$

  bringing y_k to the left hand side

$$y_k = \frac{\sum_{n=1}^N x_n p(k|n)}{\sum_{n=1}^N p(k|n)} \tag{5.9}$$

5.2.2 The Implementation of DA

We used the algorithm as suggested by Kenneth Rose [Ros98]. The idea is to start with one cluster centroid and one additional centroid. At high temperature the system tries to minimize the free energy which corresponds to the two centroids being at the same position. When the temperature is lowered, the centroids eventually split and to each of them a new perturbed centroid is assigned. Therefore, one obtains a sequential clustering solution by keeping track which cluster splits. This is depicted in Figure 5.3 and Figure 5.4. However, in practice, one can observe behavior as depicted in Figure 5.5, where this method breaks down. At very slow cooling schedules as $\alpha = 0.999$, the system is apparently in an instable state where small perturbations may result in random final states. Therefore, at each temperature, one has to check if two centroids have separated and check if one of the centroids moved to the position of another centroid. If that is the case, the system has to resetted such that each centroid has a perturbed partner again.

In our implementation, not all centroids get a perturbed partner. Instead, we randomly pick one centroid at each temperature, assign a perturbed partner and check if the solution is stable. If a unwanted configuration results, the old centroid solution is loaded and the iteration at the temperature is repeated; but this time without additional centroid. This has the advantage that the algorithm may work faster, since the positions of only $K+1$ centroids have to be computed (compared to $2K$ when each centroid gets a perturbed partner). When two centroids are at the exact same position, they behave as if they were only one centroid. This can be seen in Eqn. (5.6) since then $p(k|n) = p(l|n)$ and therefore in the M-Step that after one update, y_k and y_l are still at the same postion. A small perturbation allows the system to leave the degenerate state. When the temperature is lowered, one cluster after the other splits. When K well-defined clusters have been found, the additional perturbed centroids are removed. One should note that if the final temperature is too high, DA may not be able to find a predefined number of clusters

K.

Input: $K_{\max}$: scalar, maximal number of centroids
Input: x: (N,D)-matrix, positions in Euclidian space
Input: $T_{\max}$: scalar, starting temperature
Input: $T_{\min}$: scalar, minimal temperature
Input: t_{sep}: scalar, how far centroids are allowed to be apart before they are splitted
Input: t: scalar, parameter that defines how many iterations at each temperature are performed
Output: y: (K,D)-matrix
Initialization: $K = 1$
$y = \text{mat}[K, D]$ $y[1, :] = \frac{1}{N} \sum_{n=1}^{N} x_n$
while $T > T_{\min}$ **do**
  # select random centroid, duplicate it and perturb it by a fraction of the cluster variance
  if $K < K_{\max}$ **then**
    $K = K + 1$
    draw k randomly from $[1, \ldots, K]$
    estimate cluster variance of cluster k var=$\sum_{n=1}^{N} p[n|k] \|y_k - x_n\|^2$
    perturb y_K by a fraction $\sim \mathcal{N}(0, \rho\text{var})$
  end
  # main iteration
  while $\max_{k,d}(|y[k, d] - y_{\text{old}}[k, d]|) < t$ **do**
    $y_{\text{old}} = y$
    $p[k|n] = \frac{e^{-\frac{1}{T} d(y_k, x_n)}}{\sum_{l=1}^{K} e^{-\frac{1}{T} d(l,n)}}$
    $y[k, :] = \sum_{n=1}^{N} p[n|k] x_n$
  end
  check if cluster has splitted **if** $K < K_{\max}$ **then**
    compute KL divergence between all all $p(k|n)$. If there is one pair with KL divergence smaller than t_{sep} no cluster split occurs. Therefore repeat temperature T without additional centroids.
  end
end

Algorithm 5: The Deterministic Annealing algorithm for the K-means objective function. One starts with high temperature and one cluster. At each temperature one centroid is doubled. If in the following iteration the centroids part, the number of clusters K is increased by one. If not, the iteration is repeated without the additional centroid.

5.2.3 The Starting Temperature

One wishes to start in a configuration that is maximally non-commital. I.e., no matter how many centroids there are in the system, they should all start at the same location. This is accomplished by starting at a high temperature T, since for very high temperatures the entropy is the dominant factor in the minimization of the free energy and not the internal energy $U := \mathbb{E}[H]$). At hight T, the probability distribution for a configuration is $p(c)$ =const. For the K-means objective function, the centroids start at the center of mass. When lowering the temperature, the configuration that minimizes the free energy will eventually be one where the centroids split. Therefore, the temperature has to be just high enough to ensure a stable configuration. We need a criterion to check if the temperature is high enough. In Eqn. (5.9) we find the local optimum by taking the derivative. Possibly, we only find an extremum, i.e., the second derivative vanishs. This is when a split occurs. Given the current probability distribution $p(c)$, one can compute if a split occurs. I.e., it exists directional derivative along η such that

$$\partial_{\epsilon^2} F(y + \epsilon\eta) \leq 0 \; . \tag{5.10}$$

Performing the differentiation, one obtains the criterion that a cluster split occurs when the largest eigenvalue $\lambda_{\max}$ of a cluster's estimated covariance matrix

$$C_y = \frac{1}{N} \sum_{n=1}^{N} p(n|k)(y - x_n)(y - x_n)^T \tag{5.11}$$

satisfies

$$T_c \leq 2\lambda_{\max} \; . \tag{5.12}$$

That means, that the starting temperature must be at least $2\lambda_{\max}$.

5.2.4 The Runtime Per Iteration

As for the K-means algorithm, the runtime is $\mathcal{O}(NKD)$ per iteration, but with larger prefactors. Additionally, the structure of the algorithm makes it necessary to traverse matrices in column by column, which gives a large penalty since in C resp. Python, matrices are row major.

5.2.5 Is Deterministic Annealing Deterministic?

At each new temperature an additional centroid is added to the system at the position of an already existing centroid. To prevent degenerate states, the centroid and its brother are perturbed a little bit in a random direction. This small random perturbation leads to very different outcomes.

In Figure 5.2 one can see that our implementation (see Algorithm 5) of DA is not deterministic. And that, though it has been designed to be deterministic.

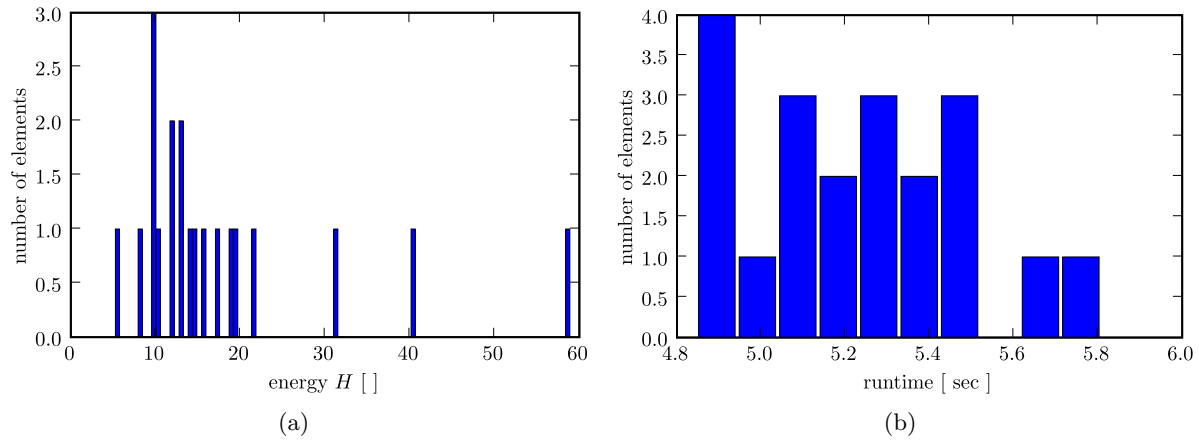


Fig. 5.2: Restarting DA on the toy dataset from Figure 8.2. One can see, that DA is by no means deterministic.

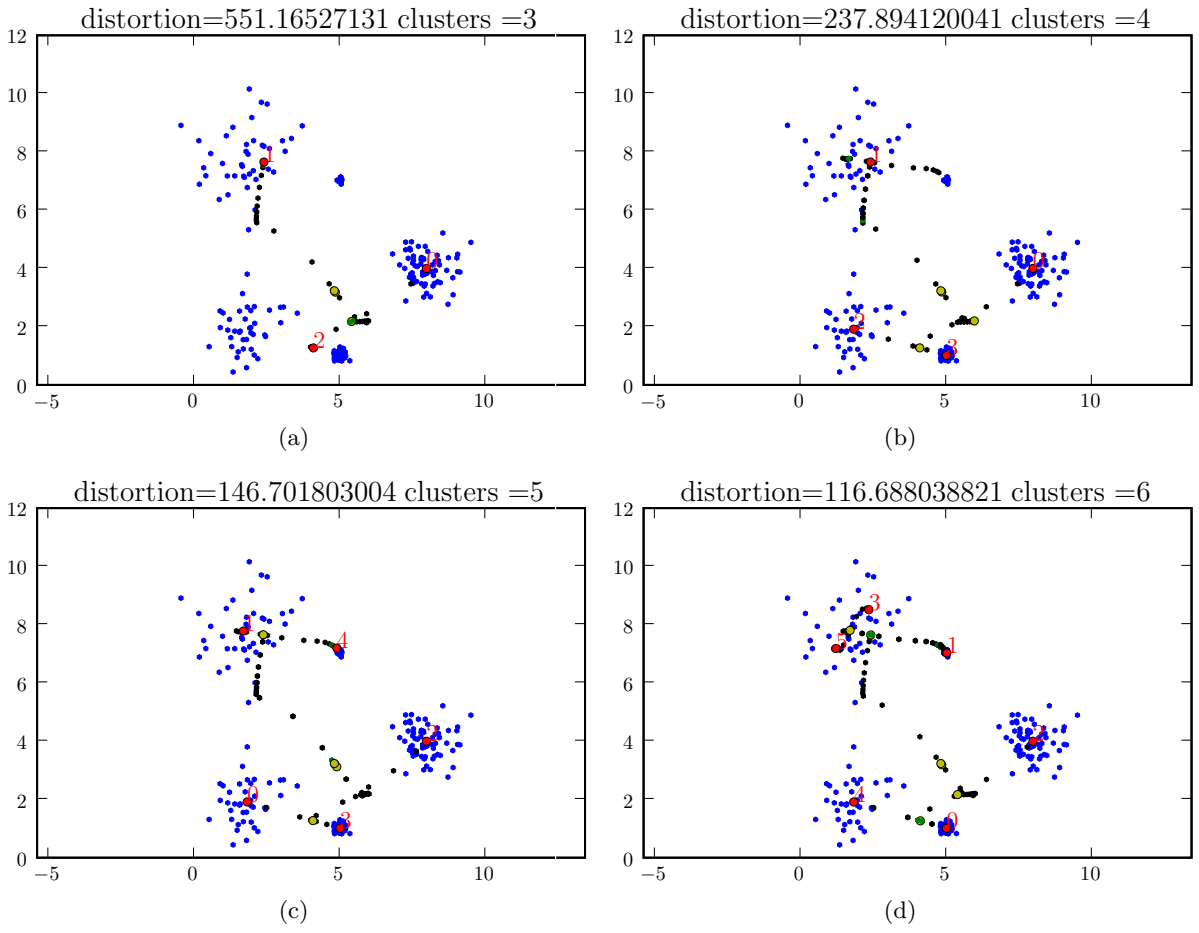


Fig. 5.3: Examples where Deterministic Annealing as suggested by Rose finds good solutions. We start DA at a sufficiently high temperature T s.t. the two initial centroids are located at the center of mass (yellow dot in the middle). Lowering the temperature changes the objective function F. At a critical temperature T^c, the two centroids seperate; each centroid gets again a perturbed partner. The location where splits occur are plotted as yellow dots. Further lowering of the temperature eventually results in additional splittings. The data points are plotted as blue dots. The red dots show the final location of the centroids at the final lowest temperature. The green dots are the position of the centroids at each temperature. The black dots are the positions of the centroids in the iteration when the temperature is fixed. One can see, that DA works sequentially since the subclusters can be attributed to the same group as the parent cluster.

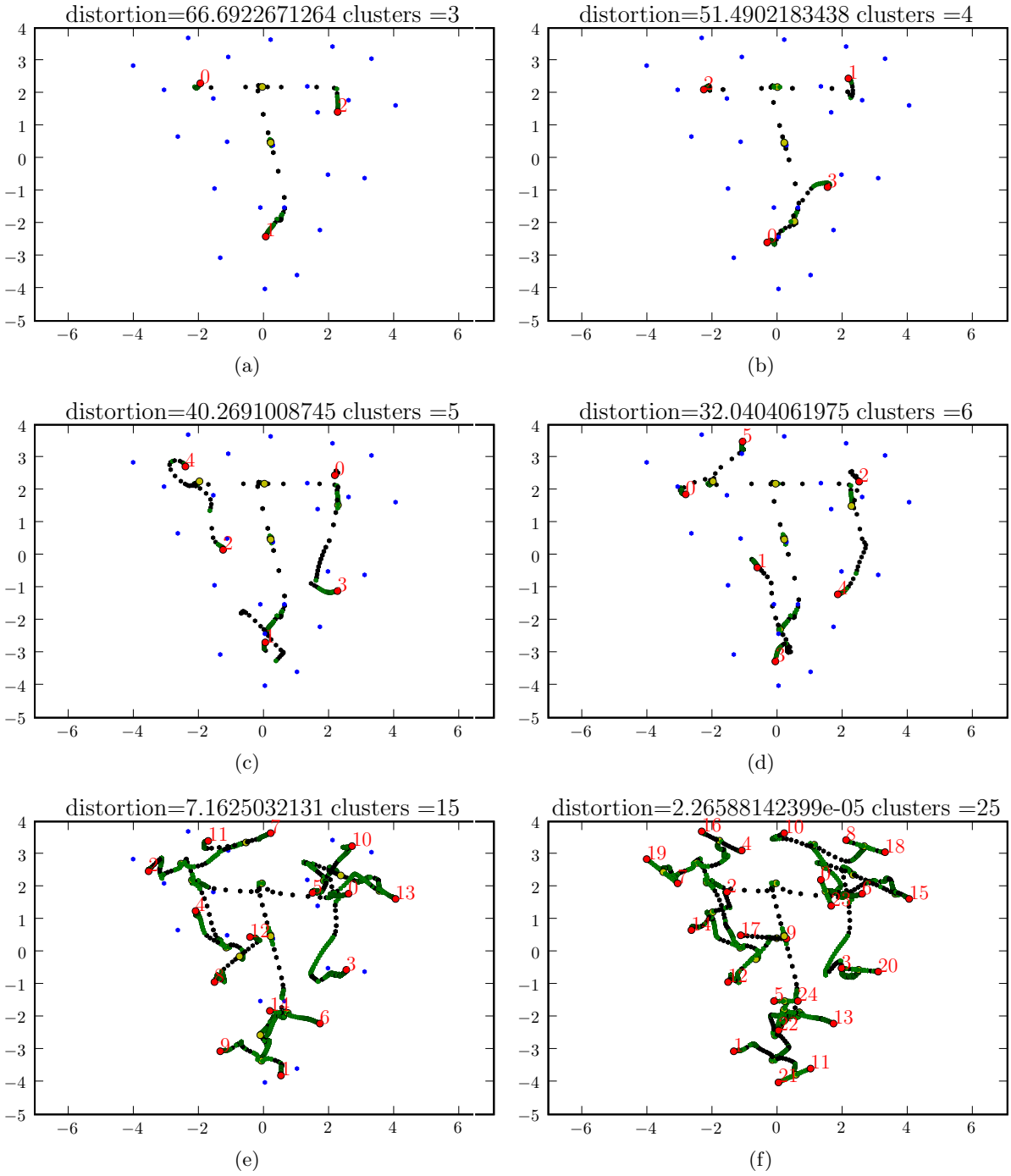
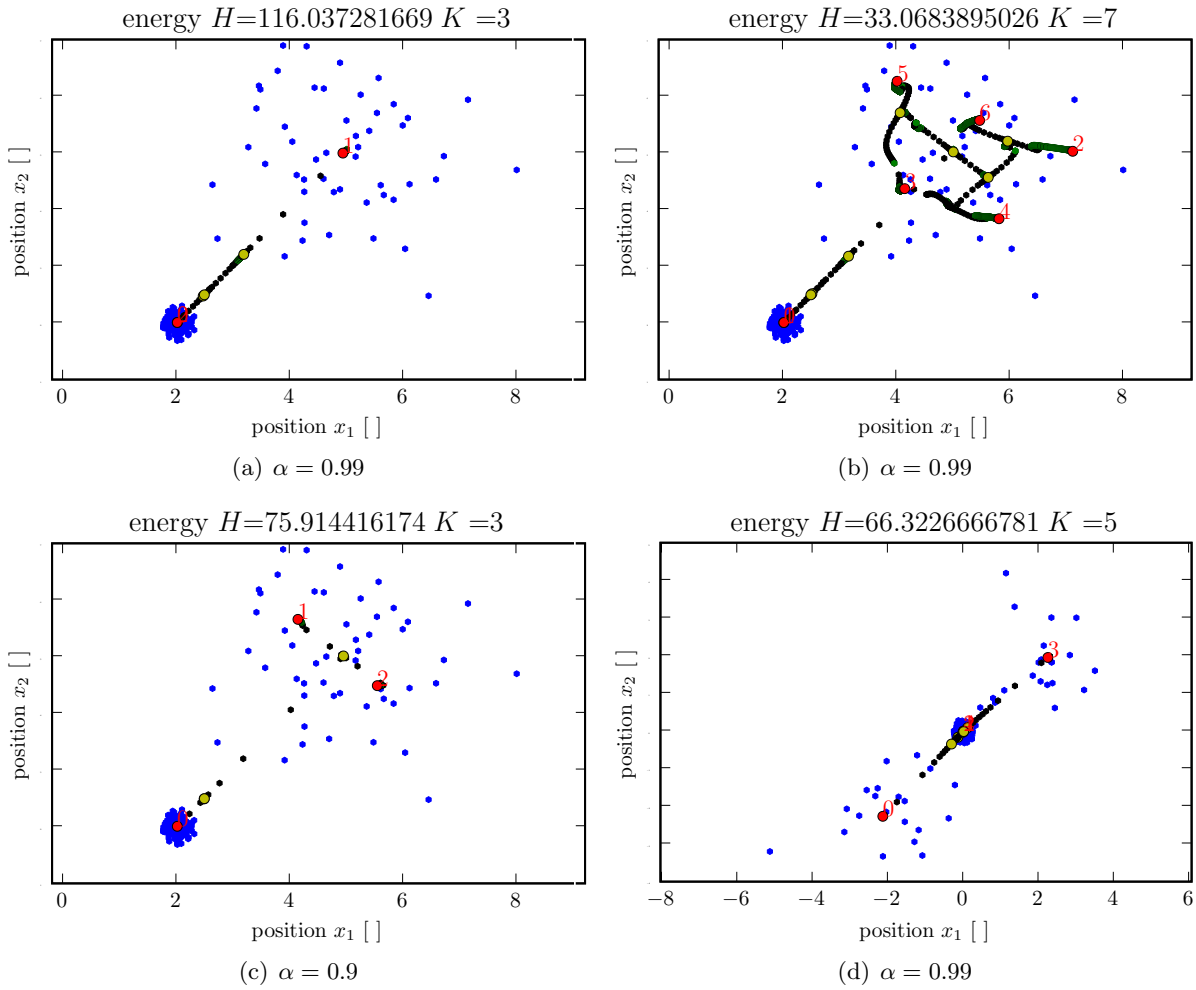


Fig. 5.4: Another example, where Deterministic Annealing, as suggested by Rose, finds a good solution. In (a) one can see, that at the centroids started at the center of mass (yellow dot). When they split, one centroid moves up, the other down (trajectory plotted as green dots). At the next critical temperature, the upper cluster splits into two subclusters (position of the split is plotted as yellow dot). Since $K = 3$ is the predefined number of clusters, the perturbed partner centroids are remove and thus no further splits are possible. (b) When the maximal number of clusters is $K = 4$, the lower cluster splits next, since it has the larger intra cluster variance. (c) to (f) show, how the clusters recursively split into subclusters. The *distortion* is another name for the objective function H.



(a) $\alpha = 0.99$ (b) $\alpha = 0.99$

(c) $\alpha = 0.9$ (d) $\alpha = 0.99$

Fig. 5.5: An example where Deterministic Annealing as suggested by Rose fails. This toy dataset has two clusters: one with high variance and low number of data points, and one cluster with five times as many data points and much lower variance. The centroids start at the center of mass (lower-left yellow dot). When they split, one centroid moves slowly in direction of the low-density cluster whereas the other moves to the center of the high-density cluster. The next cluster split occurs at the yellow dot at the edge of the low-density cluster. As predicted by Eqn. (5.11) and Eqn. (5.12), the low density cluster splits at high temperature. Apparently, the free energy F is minimized by moving one centroid exactly on the same spot where there is already another centroid. This makes sense, since at high temperature, the free energy is rather minimized by maximizing the entropy S than by minimizing the internal energy U. In other words: The cluster with low variance and many datapoints attracts perturbed centroids from the high variance cluster with few datapoints. This happens only when a slow cooling schedule as for example $\alpha = 0.99$ is used. Faster cooling (e.g. $\alpha = 0.9$) yields good solutions. The reason for that behavior is probably that a fast cooling schedule misses the the bifurcation state. That means that a small perturbation is not sufficient to move over the small energy hill. (c) The same experiment but with a faster cooling schedule $\alpha = 0.9$. Then, the result meets our expectations. (d) shows another example, where a high density cluster attracts more and more centroids (three centroids here). The fact that a high density cluster attracts centroids is reproducable; when the density is sufficiently high then almost certainly.

5.2.6 The Maximum Entropy Principle

We want to motivate the use of the Gibbs distribution. The starting point is to relax the condition that a data point belongs *either* to cluster k *or* to cluster l. Instead, a data point n is assigned a probability $p(k|n)$ that it belongs to cluster k. In other words: the configuration c of the system is assumed to be a random variable. The expected energy of the system is called the *internal energy* U:

$$U[p(c)] = \mathbb{E}_{p(c)}[H(c)] \; . \tag{5.13}$$

The big question is: which probability mass should we pick? The *maximum entropy principle* proposes to choose the probability mass function p that maximizes the *Shannon entropy* S under the constraint that p is a valid probability mass function and fixed internal energy U. Explicitly:

$$\begin{aligned} 0 \;&\stackrel{!}{=}\; \delta \left[S[p] + \beta U[p] + \mu \left(\sum_c p(c) - 1 \right) \right] \\ &= \; \delta \left[\sum_c p(c) \log p(c) + \beta \sum_c p(c) H(c) + \mu \left(\sum_c p(c) - 1 \right) \right] \\ &= \; \sum_c \underbrace{\left(\log p(c) + 1 + \beta \sum_c p(c) H(c) + \mu \right)}_{=0} \delta p(c) \; , \end{aligned} \tag{5.14}$$

where $\delta f[p] := \frac{\mathrm{d}}{\mathrm{d}\epsilon}|_{\epsilon=0} f[p + \epsilon \delta p]$. Similarly to the continuous case of the variational calculus, we search for the probability mass p which is an exremal, i.e., a small perturbation of the function f does not change its value, i.e., its variation is $\delta f = 0$. Since $\delta p(c)$ may take any value, the underbraced expression must be zero for all configurations c. This leads to $p(c) = e^{-(1+\mu)-\beta H(c)}$. Putting that in the constraint $\sum_c p(c) = 1$ one obtains $1 \stackrel{!}{=} \sum_c p(c) = e^{-(1+\mu)} \sum_c e^{-\beta H(c)}$ and therefore $\exp(1+\mu) = \sum_c \exp(-\beta H(c)) =: Z$, where Z is the so-called *partition function*. The final result reads

$$p(c) = \frac{1}{Z} \sum_c e^{-\beta H(c)} \; , \tag{5.15}$$

which is the Gibbs distribution.

5.2.7 Relation: Partition Function and Free Energy

The *free energy* F is defined as

$$F[p] = U[p] - TS[p] \; . \tag{5.16}$$

Note, that the minimization of the free energy under the constraint of fixed internal energy U is equivalent to the maximum entropy principle, i.e., $-T \max F[p] = \max S[p] - \beta U$. A short calculation shows a handy formulation of the free energy:

$$\begin{aligned} -\beta F \;&=\; S - \beta U \\ &= \; -\sum_c p(c) \left[\log p(c) + \beta H(c) \right] \\ &= \; -\sum_c p(c) \left[\log e^{-\beta H(c)} - \log Z + \beta H(c) \right] \\ &= \; \log Z \; . \end{aligned}$$

Hence, the relation between partition function and free energy is given as

$$F = -T \log Z \; . \tag{5.17}$$

All thermodynamic potentials can be written as a simple function of the partition function. In other words, the whole information about a statistical system can be described by the partition function.

6 ACM-Algorithms

The goal is to optimize the objective function Eqn. (3.11) [PHB99]. As for the K-Means objective function, one can easily derive simple iterative algorithms that correspond to the K-Means algorithm or a non-zero temperature version which corresponds to DA with a fixed temperature $T = 1$.

while *not converged* **do**
  foreach $n \in [1, N]$ **do**
    $c_n = \text{argmax}_k \sum_{d=1}^{D} h_{nd} \log q_{kd}$
  foreach $k \in [1, K]$ **do**
    $q_k = \frac{1}{|C_k|} \sum_{n=1}^{N} M_{kn} h_n$

Algorithm 6: The hard assignment ACM algorithm. $M_{kn} = \delta_{kc_n}$. h_n are the data points (histograms). q_k are the centroids.

while *not converged* **do**
  foreach $n \in [1, N]$ **do**
    $p(k|n) = \frac{1}{Z_n} \exp\left(\sum_{d=1}^{D} h_{nd} \log q_{kd}\right)$
  foreach $k \in [1, K]$ **do**
    $q_k = \sum_{n=1}^{N} p(n|k) h_n$

Algorithm 7: The soft assignment ACM algorithm. This is equivalent to deterministic annealing at temperature $T = 1$. $M_{kn} = \delta_{kc_n}$. h_n are the data points (histograms). q_k are the centroids. $p(k|n)$ are the probabilities that data point n is assigned to cluster k. The partition function is $Z_n = \sum_{k=1}^{K} \exp\left(\sum_{d=1}^{D} h_{nd} \log q_{kd}\right)$. The conditional probability $p(n|k)$ is calculated as $p(n|k) = \frac{p(k|n)}{\sum_{k=1}^{K} p(k|n)}$.

6.0.8 Derivation of the Hard Assignment ACM Algorithm

The goal is to find the minimum of the ACM objective function Eqn. (3.12). Analogously to the EM algorithms, the centroid q is treated as hidden variable, and one iterates between estimation of the current configuration and maximization of q. The following calculation shows that the configuration c uniquely defines centroids q_k. We search for

$$\begin{aligned} q^* &= \text{argmin}_q H(q|c) \\ &= \text{argmin}_q - \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd};, \end{aligned} \tag{6.1}$$

under the constraint that q is a probability mass function. We perform the variation on a continous probability space:

$$\begin{aligned} 0 &\stackrel{!}{=} \delta_k \left[- \sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \int h_n(x) \log q_k(x) \mathrm{d}x \; + \lambda \left(1 - \int q_k(x) \mathrm{d}x \right) \right] \\ &= \int - \sum_{n=1}^{N} M_{kn} \frac{h_n(x)}{q_k(x)} \delta q_k - \lambda \delta q_k(x) \mathrm{d}x \\ &= \int \left[- \sum_{n=1}^{N} M_{kn} \frac{h_n(x)}{q_k(x)} - \lambda \right] \delta q_k(x) \mathrm{d}x \;, \end{aligned} \tag{6.2}$$

where the differential operator δ_k is defined as

$$\delta_k F[q_1(x), \ldots, q_k(x), \ldots, q_K(x)] := \frac{\mathrm{d}}{\mathrm{d}\epsilon}|_{\epsilon=0} F[q_1(x), \ldots, q_k(x) + \epsilon\delta q_k(x), \ldots, q_K(x)] \; . \tag{6.3}$$

$\delta q_k(x)$ is the variation by which $q_k(x)$ is perturbed. $q_k(x)$ is assumed to be positive. The fundamental lemma of variations states that the expression in the brackets must vanish. Therefore

$$0 \stackrel{!}{=} \sum_{n=1}^{N} M_{kn} \frac{h_n(x)}{q_k(x)} + \lambda \; .$$

Solving for $q_k(x)$ and putting the expression in the contraint equation gives us $\lambda = -\frac{1}{|C_k|}$. Hence the only extremal is

$$q_k(x) = \frac{1}{|C_k|} \sum_{n=1}^{N} M_{kn} h_n(x) \; . \tag{6.4}$$

One would have to check higher orders of the variation and check if that is really a minimum.

6.0.9 Derivation of the Soft Assignment ACM Algorithm

The objective function is the free energy

$$\begin{aligned} F &= -T \log Z \\ &= -T \log \sum_c e^{-\beta H^{\mathrm{ACM}}(c)} \end{aligned} \tag{6.5}$$

where $H^{\mathrm{ACM}}(c) = -\sum_{k=1}^{K} \sum_{n=1}^{N} M_{kn} \sum_{d=1}^{D} h_{nd} \log q_{kd}$. This optimization problem is equivalent to a density estimation:

$$\begin{aligned} p(c) &= \frac{1}{Z} \exp\left(\frac{1}{T} \sum_{n=1}^{N} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right) \\ &= \frac{\exp\left(\frac{1}{T} \sum_{n=1}^{N} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)}{\sum_{c_1,\ldots,c_N} \prod_{n=1}^{N} \exp\left(\frac{1}{T} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)} \\ &\stackrel{(*)}{=} \frac{\prod_{n=1}^{N} \exp\left(\frac{1}{T} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)}{\prod_{n=1}^{N} \sum_{c_n} \exp\left(\frac{1}{T} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)} \\ &\stackrel{(\dagger)}{=} \prod_{n=1}^{N} \frac{\exp\left(\frac{1}{T} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)}{\sum_{c_n} \exp\left(\frac{1}{T} \sum_{d=1}^{D} h_{nd} \log q_{c_n d} \right)} \; . \end{aligned}$$

In $(*)$ we have used the trick

$$\begin{aligned} \sum_{c_1=1}^{K} \cdots \sum_{c_N=1}^{K} f_1(c_1) \cdot \cdots \cdot f_N(c_N) &= \sum_{c_1=1}^{K} \cdots \sum_{c_{N-1}=1}^{K} f_1(c_1) \cdot \cdots \cdot f_{N-1}(c_{N-1}) \cdot \sum_{c_N=1}^{K} f_N(c_N) \\ &= \prod_{n=1}^{N} \sum_{c_n=1}^{K} f_n(c_n) \; . \end{aligned} \tag{6.6}$$

In $(\dagger)$ on can see that the probability distribution factorizes if the q_k were fixed, i.e., wouldn't depend on c.

As the following derivation shows, the centroids $q_k(c_1, \ldots, c_N)$ depend uniquely on the configuration c. If q_k was known, the joint distribution would factorize and the computation would

be much easier. This naturally leads to the iterative EM algorithm, where in the E-Step the probabilities $p(k|n)$ are estimated under the assumption that q is fixed:

$$p(k|n) = \frac{\exp\left(\frac{1}{T}\sum_{d=1}^{D} h_{nd} \log q_{kd}\right)}{\sum_{l=1}^{K} \exp\left(\frac{1}{T}\sum_{d=1}^{D} h_{nd} \log q_{ld}\right)} \,. \tag{6.7}$$

We show now, that when c is fixed, then there exist unique q_k that minimize the free energy under the constraint that $\sum_{d=1}^{D} q_{kd} = 1$. We do the derivation for continuous probabilities:

$$\begin{aligned}
0 \;\stackrel{!}{=}\; & \delta_k \left[F[q] + \lambda \left(1 - \int q_k(x) \mathrm{d}x \right) \right] \\
= \;& \delta_k \left[-T \log Z + \lambda \left(1 - \int q_k(x) \mathrm{d}x \; \right) \right] \\
= \;& \delta_k \left[-T \log \prod_{n=1}^{N} \sum_{c_n} \exp\left(\frac{1}{T} \int \mathrm{d}x \; h_n(x) \log q_{c_n}(x) \right) + \lambda \left(1 - \int q_k(x) \mathrm{d}x \; \right) \right] \\
= \;& -T \sum_{n=1}^{N} \delta_k \log \sum_{c_n} \exp\left(\frac{1}{T} \int h_n(x) \log q_{c_n}(x) \mathrm{d}x \; \right) - \lambda \int \delta q_k(x) \mathrm{d}x \\
= \;& -T \sum_{n=1}^{N} \underbrace{\frac{\exp\left(\frac{1}{T}\int h_n(x) \log q_k(x) \mathrm{d}x \; \right)}{\sum_{l=1}^{K} \exp\left(\frac{1}{T}\int \mathrm{d}x \; h_n(x) \log q_l(x)\right)}}_{=p(k|n)} \frac{1}{T} \int \frac{h_n(x)}{q_k(x)} \delta q_k(x) \mathrm{d}x \; - \lambda \int \delta q_k(x) \mathrm{d}x \\
= \;& -\int \left(\sum_{n=1}^{N} p(k|n) \frac{h_n(x)}{q_k(x)} - \lambda \right) \delta q_k(x) \mathrm{d}x \;\; .
\end{aligned} \tag{6.8}$$

Therefore, $q_k(x) = \frac{1}{\lambda} \sum_{n=1}^{N} p(k|n) h_n(x)$. The Lagrange parameter λ can be obtained by putting the expression in the constraint function, i.e.,

$$\begin{aligned}
1 \;\stackrel{!}{=}\; & \int q_k(x) \mathrm{d}x \\
= \;& \frac{1}{\lambda} \int \mathrm{d}x \; \sum_{n=1}^{N} p(k|n) h_n(x) \\
= \;& \frac{1}{\lambda} \sum_{n=1}^{N} p(k|n) \int \mathrm{d}x \; h_n(x) \\
= \;& \frac{1}{\lambda} \sum_{n=1}^{N} p(k|n)
\end{aligned}$$

We obtain as extremal

$$q_k(x) = \frac{\sum_{n=1}^{N} p(k|n) h_n(x)}{\sum_{n=1}^{N} p(k|n)} \,. \tag{6.9}$$

7 Affinity Propagation (AP)

The final goal of this section is the derivation of the affinity propagation (AP) algorithm as a special case of the sum-product algorithm. The tutorial paper "Factor Graphs and the Sum-Product Algorithm" by Kschischang [KFL01] takes the following approach: At first the *marginalize a product function'* (MPF) problem is stated. After that, it is shown how the factorization of the product function can be represented as factor graph. Then the sum-product algorithm is introduced: it solves the "marginalize a product function" problem exactly on trees by sending messages between nodes of the factor graph. Only after that, generalizations of the sum-product algorithm are considered, i.e., the application of the sum-product algorithm on loopy graphs and the application of the sum-product algorithm in other semirings.

As we will see, AP is the sum-product algorithm in the min-sum semiring on a loopy factor graph. Therefore, we take the opposite approach as Kschischang and start with the most general form of the sum-product algorithm. The sum-product algorithm works on a variety of commutative semirings (Tab. 7.1): we show that applying the sum-product algorithm on each of the semirings solves a different interesting problem, as for example the MPF problem, the *satisfiability problem* or the *optimize a sum-function* (OSF) problem.

7.1 Some Mathematical Preliminaries

We want to show later for what kind of problem the sum-product algorithm can be applied in principal. To do so, we need to make the following definition first: A *commutative semiring* is a set $\mathbb{Y}$ equipped with two binary operations $\oplus : \mathbb{Y} \times \mathbb{Y} \to \mathbb{Y}$ and $\odot : \mathbb{Y} \times \mathbb{Y} \to \mathbb{Y}$ such that the following axioms hold:

1. $(\mathbb{Y}, \oplus)$ is a commutative monoid with identity element $\bar{0}$.
2. $(\mathbb{Y}, \odot)$ is a commutative monoid with identity element $\bar{1}$.
3. The distributive law $y_1 \odot (y_2 \oplus y_3) = y_1 \odot y_2 \oplus y_1 \odot y_3$ holds.

Examples are the field of real numbers with the usual multiplication and addition operator $(\mathbb{R}, +, \cdot)$ or the binary field $(\mathbb{Z}_2, +, \cdot)$. Note, that the application of the distributive law can save arithmetic operations: two on the left hand side and three on the right hand side. For a detailed definition of group and monoid see Appendix B.

Later, the $\oplus$ operator defines what problem the sum-product algorithm solves. For example it is the max operator for the OSF or + for the MPF. The $\odot$ is always the operator that defines the structure of a function, e.g. the multiplication between factors of a joint probability mass function.

7.2 Factor Graphs

In the following we will solely focus on functions f that have the properties that the *codomain is a commutative semiring, the domain is finite and the function factorizes*! Explicitly:

$$\begin{aligned} f : (\mathbb{X}, +, \cdot) &\longrightarrow (\mathbb{Y}, \oplus, \odot) \\ x &\longmapsto f(x) \,, \end{aligned} \tag{7.1}$$

where $\mathbb{X}$ is the domain and $\mathbb{Y}$ the codomain. $(\mathbb{Y}, \oplus, \odot)$ is a commutative semiring. A necessary condition that the sum-product algorithm can be applied is that the set $\mathbb{X}$ is finite. The factorization on the codomain is given by

$$f(x) = f(x_1, \ldots, x_N) = \bigodot_{j=1}^{J} f_j(x_j) = f_1(x_{j_1}) \odot \cdots \odot f_J(x_{j_J}) \,, \tag{7.2}$$

where $x = (x_1, \ldots, x_N)$ and $x_j = (x_{j_1}, \ldots, x_{j_{N_j}})$ is a subset of x. Among other possibilities, such a factorization can be represented graphically as a factor graph.

A *factor graph* is an undirected, bipartite graph that represents the factorial structure of a function (Eqn. (7.2)) with two different kinds of nodes: *variable nodes* x_n and *factor nodes* f_j. A variable node x_n is connected to the factor node f_j iff x_n is an argument of f_j.

7.3 The Sum-Product Algorithm

In Algorithm 8 the *sum-product* algorithm on loopy factor graphs is explained. The motivation is nicely explained in [AM00, KFL01]. Here, we state the update rule and show only later a simple example how it works. When the factor graph is a tree, the sum product algorithm stops when all messages have traversed the whole tree.

while *not converged* **do**
  (1) variable node x_n to factor node f_j: $\mu_{x_n \to f_j}(x_n) = \bigodot_{g \in \mathcal{N}(x_n) \setminus f_j} \mu_{g \to x_n}(x_n)$
  (2) factor node f_j to variable node x_n:
  $\mu_{f_j \to x_n}(x_n) = \bigoplus_{\sim x_n} \left(f_j(x_j) \odot \bigodot_{y \in \mathcal{N}(f_j) \setminus x_n} \mu_{y \to f_j}(y) \right)$
end

Algorithm 8: The generalized sum-product algorithm. $\mathcal{N}(x)$ is defined as the set of all neighbors of node x, $\mathcal{N}(x) \setminus f$ are all adjacent of node x nodes but f. The products $\bigodot a_i = a_1 \odot a_2 \odot \cdots \odot a_J$ and $\bigoplus a_i = a_1 \oplus a_2 \oplus \cdots \oplus a_J$ are generalized products and sums. If x_n or f_j are in the subscript they stand for a node in the factor graph.

7.4 The Sum-Product Algorithm in Different Semirings

In Table 7.1 one can see a collection of semirings. One can plug in any function that factorizes, has a finite domain and a codomain that is a commutative semiring and simply run the sum-product algorithm.

1. In the case that the semiring is the ordinary field of real numbers $(\mathbb{R}, +, \cdot)$ the sum-product algorithm tries to solve the *marginalize a product function*, i.e., find

$$p(x_n) = \sum_{\sim x_n} p(x_1, \ldots, x_N) = \sum_{\sim x_n} \prod_{j=1}^{J} p_j(x_j) \;, \tag{7.3}$$

   for all $n \in [1, \ldots, N]$. The sum $\sum_{\sim x_n}$ is called the *not-sum* and sums over all possible values of x_i but x_n. In that case, the sum-product algorithm on trees is also known as *belief propagation* (BP) resp. *loopy belief propagation* when the factor graph has loops.

2. In the case that the semiring is $(\mathbb{R}, OR, AND)$ (number 9 in Table 7.1) the sum-product algorithm tries to solve the *SAT problem*, i.e. find out if there exists a configuration of Boolean variables that satisfies

$$E \;=\; \bigvee_{x_1, \ldots, x_N \in \{0,1\}} \bigwedge_{j=1}^{J} b_j(x_j) \tag{7.4}$$

   where $\wedge$ is the AND operator, $\vee$ the OR operator and $b_j(x_j) = b(x_{j_1} \ldots x_{j_K}) \in 0, 1$.

3. Finally, if the semiring is $(\mathbb{R}, \min, +)$ (number 7 in Table 7.1) , the problem that the sum-product algorithm tries to solve is the *optimize a sum-function* problem, i.e.,

$$f(x^*) \;=\; \min_x f(x) = \min_x \sum_{j=1}^{J} f_j(x_j) \;, \tag{7.5}$$

   where $x^* = \text{argmin}_x f(x)$, $x = (x_1, \ldots, x_N)$ and x_j as above.

Note, that the argmin_x is the complete not-sum over all variables.

	semirings				sum-product algorithm	
	K	$(\oplus, \bar{0})$	$(\odot, \bar{1})$	name	problem	famous algorithm
1.	A	$(+, 0)$	$(\cdot, 1)$			
2.	A[x]	$(+, 0)$	$(\cdot, 1)$			
3.	$A[x, y, \dots]$	$(+, 0)$	$(\cdot, 1)$			
4.	$[0, \infty)$	$(+, 0)$	$(\cdot, 1)$	sum-product	MPF	Belief Propagation
5.	$(0, \infty]$	$(\min, \infty)$	$(\cdot, 1)$	min-product		
6.	$(0, \infty]$	$(\max, 0)$	$(\cdot, 1)$	max-product	max. a pf	Viterbi Algorithm
7.	$(-\infty, \infty]$	$(\min, \infty)$	$(+, 0)$	min-sum	min. a sf	Affinity Propagation
8.	$[-\infty, \infty)$	$(\max, -\infty)$	$(+, 0)$	max-sum	max. a sf	Affinity Propagation
9.	$\{0, 1\}$	$(OR, 0)$	$(AND, 1)$	Boolean	SAT problem	
10.	2^S	$(\cup, \varnothing)$	$(\cap, S)$			
11.	Λ	$(\vee, 0)$	$(\wedge, 1)$			
12.	Λ	$(\wedge, 1)$	$(\vee, 0)$			

Tab. 7.1: We abbreviated product-function as pf and sum-function as sf. In the left part various commutative semi-rings are assembled that can be used with the sum-product algorithm. On the right side we filled in what problem the sum-product algorithm tries solves in the semirings as well as the most famous algorithm.

7.5 The Termination of the Iteration

The iteration of the sum-product algorithm is the same for all semirings. However, after convergence, the last step differs.

7.5.1 Termination of Belief Propagation

In the case of BP, the goal is the estimate of the marginal probability of $p(x_n)$ for all n. This is done by taking the product of all incoming messages from adjacent factor nodes, i.e.

$$p(x_n) = \prod_{h \in \mathcal{N}(x_n)} \mu_{h \to x_n}(x_n) \; . \tag{7.6}$$

In the case when the graph has loops, one referes to the BP approximations of the marginal probabilities as *beliefs* $b_n(x)$ to stress the fact that they are not the exact marginals $p_n(x)$

7.5.2 Termination of the Sum-Product algorithm in the Min-Sum Semiring

As explained in the previous section, the goal is the optimization of the objective function. This is done as follows:

$$x_n^* = \operatorname{argmin}_{x_n} \sum_{h \in \mathcal{N}(x_n)} \mu_{h \to x_n}(x_n) \tag{7.7}$$

$$H(x^*) = \min_{x_n} \sum_{h \in \mathcal{N}(x_n)} \mu_{h \to x_n}(x_n) \; . \tag{7.8}$$

Notice that for the computation of a extremum of a function, only the incoming messages at one variable node has to be known. To know the configuration, one needs incoming messages at all variable nodes.

7.6 Is the Sum-Product Algorithm Exact?

When the factor graph is a tree, the sum-product algorithm returns the exact result. The usual schedule is to start sending messages from the leaves. When all messages have traversed the tree, the algorithm terminates.

When the factor graph is not a tree, i.e. the factor graph has loops, there is theoretically little known how the result should be interpreted. However, there is experimental evidence that the sum-product algorithm on loopy factor graphs often converges to good solutions. An example, where the sum-product algorithm on loopy factor graphs gives record breaking results are error correcting codes [Mac99].

7.7 The Message Passing Schedule

An interesting problem is the scheduling of the messages. This is not of particular interests if the graph is a tree but has importance on loopy graphs. For instance, since either the messages $\mu_{f_j \to x_n}$ or $\mu_{x_n \to f_j}$ are updated first, one has to make a copy of the messages that are updated first. This method is called *batch processing*. However, saving all messages at each step may need too much memory. A typical approach is to use some kind of leap-frog approach (as it is used when the equations of motion are integrated), where the computation of one kind of message uses the messages of the following iteration step. This is called a *sequential* update rule. To compensate for possible oscillations one can introduce a *damping factor* λ: $\mu^{\text{new}} = (1-\lambda)\mu^{\text{new}} + \lambda\mu^{\text{old}}$.

7.8 Explicit Example

We show a simple example how the sum-product algorithm in the min-sum semiring can be used to optimize a function. This example works on a tree, which means that the sum-product algorithm will give the correct result. We want to solve the following optimization problem:

$$f(x_1^*, x_2^*) = \max_{x_1 x_2} f_1(x_1) \odot f_2(x_2) = \max_{x_1} \left\{ f_1(x_1) + \max_{x_2} f_2(x_1, x_2) \right\} = \max_{x_1} \left\{ x_1 + \max_{x_2} [x_1 \neq x_2 \right\} , \tag{7.9}$$

where the function f is defined as $f : (\mathbb{Z}_2^2, (+,0), (\cdot,1)) \to (\mathbb{R}, (\min, \infty), (+,0))$. $\mathbb{Z}_2$ is the field of binary operators with $x \in \{0,1\}$ and the usual binary operators $1+1=0$. We use Iverson's notation $[\text{true}] = 1$ and $[\text{false}] = 0$. The expression's factor graph is depicted in Figure 7.1.

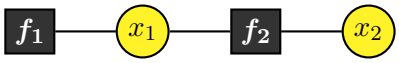


Fig. 7.1: The factor graph of Eqn. (7.9).

The solution is $(x_1^*, x_2^*) = (1,0)$ and $f(x_1^*, x_2^*) = 2$. The way to compute it using factor graphs is the following: Start sending messages from the leaves and stop when the whole tree has been traversed. We use the notation of $\mathbf{x}$ when we consider the all the values that the variable x can take. We write $\mu_{f \to x}(\mathbf{x})$ and we mean the set of messages that have to be sent. Explicitly $\mu_{f \to x}(\mathbf{x}) = \{\mu_{f \to x}(x=0), \mu_{f \to x}(x=1)\}$.

Step 1 $\mu_{f_1\to x_1}(\mathbf{x}_1) = \max_{\sim \mathbf{x}_1} f_1(\mathbf{x}_1) = f_1(\mathbf{x}_1) = \{f_1(x=0), f_1(x=1)\} = \{1, 0\}$
$\mu_{x_2\to f_2}(\mathbf{x}_2) = 0$
Step 2 $\mu_{x_1\to f_2}(\mathbf{x}_1) = f_1(\mathbf{x}_1)$
$\mu_{f_2\to x_1}(\mathbf{x}_1) = \max_{\sim \mathbf{x}_1} f_2(\mathbf{x}_1, \mathbf{x}_2) = \max_{\mathbf{x}_2} f_2(\mathbf{x}_1, \mathbf{x}_2)$
$= \max(\mathbf{x}_1 \neq 0, \mathbf{x}_1 \neq 1) = \{\max_y f_2(x_1 = 0, x_2), \max_y f_2(x_1 = 1, x_2)\}$
$= \{\underbrace{1}_{x=0}, \underbrace{1}_{x=1}\}$
Step 3 $\mu_{x_1\to f_1}(\mathbf{x}_1) = \mu_{f_2\to x_1}(\mathbf{x}_1) = \{1, 1\}$
$\mu_{f_2\to x_2}(\mathbf{x}_2) = \max_x [\{f_2(x_1, x_2 = 0), f_2(x_1, x_2 = 1\} + \mu_{x_1\to f_2}(\mathbf{x}_1)]$
$= \{\max[f_2(x = 0, y = 0) + 0, f_2(x = 1, y = 0) + 1], \max[f_2(x_1 = 0, x_2 = 1) + 0, f_2(x_1$
$= \{2, 1\}$
Step 4 Terminating the message passing.
$x^* = \text{argmax}_{x_1} [\mu_{f_1\to x_1}(x_1) + \mu_{f_2\to x_1}(x_1)]$
$= \text{argmax}(\underbrace{0+1}_{\text{with } x_1=0}, \underbrace{1+1}_{\text{with } x_1=1}) = 1$
$\max f(x_1, x_2) = 2$
$x_2^* = \text{argmax}_{x_2} \mu_{f_2\to x_2}(x_2) = 0$
$\max f(x_1, x_2) = 2$

If only the extremum of the function is of interest it suffices to make the factor graph rooted with a variable node as root. Then send messages from the leaves to the root. When all messages have arrived at the root, compute the extremum of the function by applying Eqn. (7.8).

7.9 Derivation of the Affinity Propagation Algorithm

We are now in place to derive the affinity propagation algorithm. The objective function is given by

$$H(e) = -\sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{m=1}^{N} \delta_m(e) \,, \tag{7.10}$$

$$\delta_m(e) = \begin{cases} \infty & e_m \neq m \text{ and } e_n = m \text{ for a } n \\ 0 & \text{else} \end{cases} \tag{7.11}$$

$$e^* = \text{argmin}_{e\in\mathbf{e}} H(e) \tag{7.12}$$

where $e = (e_1, \ldots, e_N)$ and $e_n \in \{1, \ldots, N\}$ is the exemplar label that assigns each data point x_n another data point x_{e_n} as exemplar. As above we use bold notation for set of all possible values e_n can take, i.e., $\mathbf{e} = (\mathbf{e}_1, \ldots, \mathbf{e}_N)$ and $\mathbf{e}_n = \{1, 2, \ldots, N\}$ the . The function $\delta_m(e)$ is a function that guarantees that only valid configurations are possible. There are many other possible choices for the formulation of the constraint. The alternatives that we have tried out led to complicated expressions that we couldn't simplify. For example we have used $\delta_m(e) = \{\infty \text{ if } \delta_m = n \ \wedge \ \delta_n \neq n, \quad 0 \text{ else }\}$. From Table 7.1 we see that the commutative semiring that we can use is the min-sum semiring (number 7), i.e. we have to solve the min-sum problem. For this commutative semiring the generalized sum-product algorithm is given by

while *not converged* **do**
  variable node x_n to factor node f_j: $\mu_{x_n\to f_j}(x_n) = \sum_{g\in\mathcal{N}(x_n)\setminus f_j} \mu_{g\to x_n}(x_n)$
  factor node f_j to variable node x_n:
  $\mu_{f_j\to x_n}(x_n) = \min_{\sim x_n} \left(f_j(x_j) + \sum_{y\in\mathcal{N}(f_j)\setminus x_n} \mu_{y\to f_j}(y)\right)$
end

Algorithm 9: The min-sum algorithm. Observe, that for each possible x_n a unique message $\mu_{f_j\to x_n}(x_n)$ has to be sent.

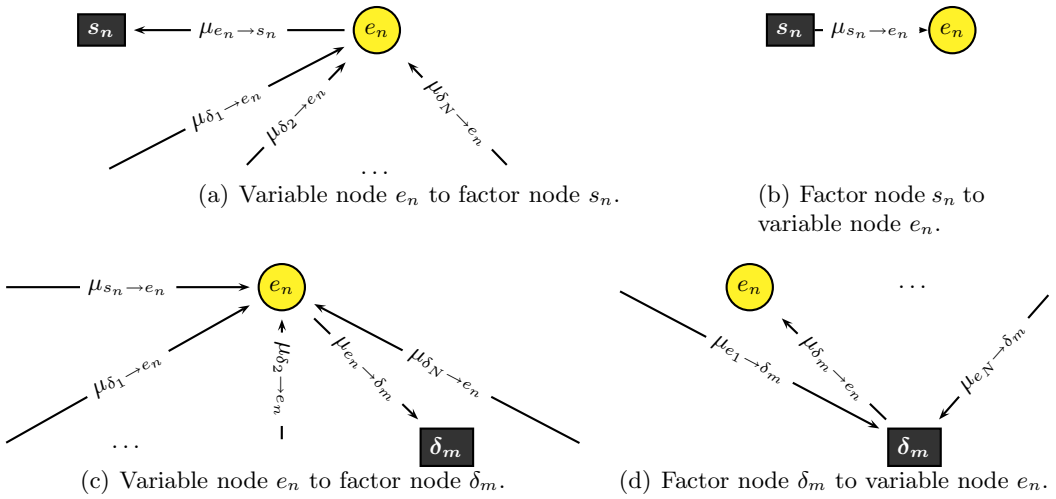
Applied on the factor graph of Figure 7.3 we need to send four kinds of messages:

1. **variable node e_n to factor node s_n:**
   $\mu_{e_n \to s_n}(\mathbf{e}_n) = \sum_{f \in \mathcal{N}(e_n) \setminus s_n} \mu_{f \to e_n}(\mathbf{e}_n)$.
   These messages do not have to be considered because the are never used.

2. **factor node s_n to variable node e_n:**
   $\mu_{s_n \to e_n}(\mathbf{e}_n) = \min_{\sim \mathbf{e}_n} s_n(\mathbf{e}_n) = s_n(\mathbf{e}_n)$.

3. **variable node e_n to factor node δ_m:**
   $\mu_{e_n \to \delta_m}(\mathbf{e}_n) = \mu_{s_{e_n} \to e_n}(\mathbf{e}_n) + \sum_{l \neq m} \mu_{\delta_l \to e_n}(\mathbf{e}_n)$.

4. **factor node δ_m to variable node e_n:**
   $\mu_{\delta_m \to e_n}(\mathbf{e}_n) = \min_{\sim \mathbf{e}_n} \left[\delta_m(\mathbf{e}) + \sum_{l \neq n} \mu_{e_l \to \delta_m}(\mathbf{e}_l) \right]$.



(a) Variable node e_n to factor node s_n.

(b) Factor node s_n to variable node e_n.

(c) Variable node e_n to factor node δ_m.

(d) Factor node δ_m to variable node e_n.

Fig. 7.2: This is a graphical representation of how the sum-product algorithm computes the messages that are sent from node to node. For example, in (a) the message $\mu_{e_n \to \delta_{nm}}$ is computed as sum of the incoming messages $\mu_{\delta_{nl} \to e_m}$. Each message consists of N possible values that the configuration e_n can assume.

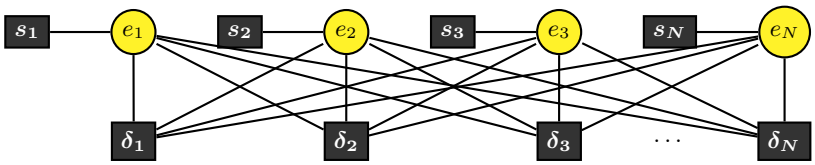


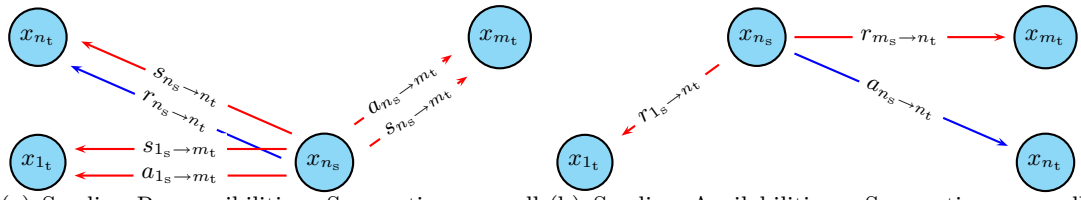
Fig. 7.3: The affinity propagation factor graph. This graph is a graphical representation of the factorization of the objective function Eqn. (7.10). The factor nodes are connected to the variable nodes iff it is a variable of the function. The factor nodes $s_n = s(x_{e_n}, x_n)$ hold the actual input to the algorithm, since they compute the distance between datapoint x_n and its exemplar x_{e_n}. The δ-functions take as input the exemplar labels of all variable nodes and check if it is a valid configuration. From the graph one can also make a rough estimate of the computational complexity: At each iteration, messages have to be sent from each factor node to each connected variable node taking as input all incoming messages. That means, if no tricks and optimizations are applied, one needs to send $\mathcal{O}(N^3)$ messages. Each message consists of all possible values a variable node can take, i.e., in our case N values. This means that each iteration needs $\mathcal{O}(N^4)$ arithmetic operations. Frey managed to get it down to $O(N^2)$ per iteration which is a considerable speedup.

7.10 The Final Version of the Affinity Propagation Algorithm

Here we present the final form of affinity propagation as it can be found in [FD07a]. Each iteration can be done in $\mathcal{O}(N^2)$ operations. Contrary to the sum-product algorithm which is

hard to grasp, affinity propagation allows an intuitive explanation of the transmitted messages. The *responsibility* sent from node x_{n_s} to x_{n_t} reflects the accumulated evidence how appropriate it would be for node x_{n_s} to have x_{n_t} as exemplar. The *availability* sent from node x_{n_s} to x_{n_t} indicates the accumulated evidence that node x_{n_t} should choose x_{n_s} as exemplar.

Attention should be payed to the fact that the algorithm works only on outgoing messages though outgoing messages are actually computed by considering all incoming messages. Therefore one does not have to implement the algorithm on a bidirectional graph but can use an edge list graph.



(a) Sending Responsibilities. Summation over all "incoming " edges (red), also the self-loops.

(b) Sending Availabilities. Summation over all "incoming" edges (red) but self-loops.

Fig. 7.4: The update of availabilities and responsibilities. One outgoing message is computed by all incoming messages. However, the direct implementation with incoming and outgoing messages leads to an implementation with $\mathcal{O}(N^3)$ arithmetic operations per iteration. With the trick that all messages are outgoing messages one can implement it in $\mathcal{O}(N^2)$ arithmetic operations. This is a graphical representation how at each step the algorithm computes the new availability and responsibility messages. The directions the arrows are pointing at correspond to the actual implementation. The messages that are actually incoming messages are plotted in red.

Input:
similarities $s_{n_s \to n_t}$
preferences $s_{n_s \to n_s}$
Initialization:
set $a_{n_s \to n_t} = r_{n_s \to n_t} = 0$ for all n_s and $n_t \in [1, \ldots, N]$
Iteration:
while *not converged* **do**
  update responsibilities
  $r_{n_s \to n_t} \overset{\lambda}{=} s_{n_s \to n_t} - \max_{m_t \neq n_t} [a_{n_s \to m_t} + s_{n_s \to m_t}]$
  update availabilities
  $a_{n_s \to n_t} \overset{\lambda}{=} \min \left\{ 0, r_{n_t \to n_t} + \sum_{m_s \neq \{n_s, n_t\}} \max [0, r_{m_s \to n_t}] \right\}$
  $a_{n_t \to n_t} \overset{\lambda}{=} \sum_{m_s \neq n_t} [0, r_{m_s \to n_t}]$

Algorithm 10: The Affinity Propagation algorithm. We use here the notation a for the availability, r for the responsibility and s for the similarity. They are messages that are sent between nodes. For example $a_{n_s \to n_t}$ is the availability sent from a source node x_{n_s} to the target node x_{n_t}. The notation $\overset{\lambda}{=}$ means $a^{\text{new}} = (1-\lambda)a^{\text{new}} + \lambda a^{\text{old}}$; it prevents oscillations. The naive implementation of this algorithm needs $\mathcal{O}(N^3)$ arithmetic operations per iteration. A factor $\mathcal{O}(N)$ can be gained by computing the unrestricted sums, seperately saving the special cases when a restriction is active and finally computing the value of the restricted sum by combining the previously computed results.

7.10.1 The Termination of Affinity Propagation

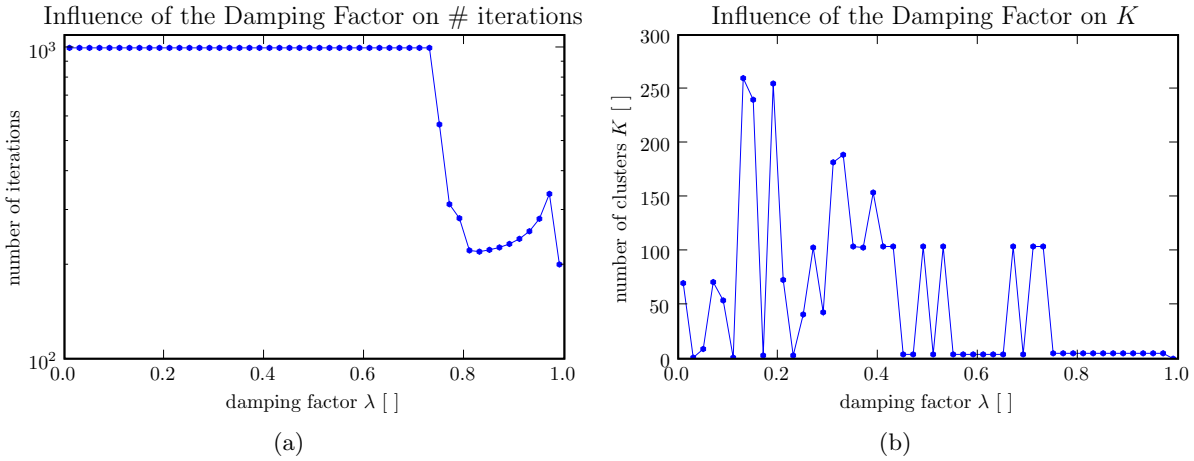
Analogously to the termination of the sum-product algorithm in the min-sum semiring, the estimated exemplar labels e_n are computed by

$$e_n^* = \text{argmin}_{e_n} \left[\sum_{h \in \mathcal{N}(e_n)} \mu_{h \to e_n}(e_n) \right] \tag{7.13}$$

$$= \text{argmin}_{e_n} \left[a_{n_s \to n_t} + s_{n_s \to n_t} \right] . \tag{7.14}$$

7.10.2 The Influence of the Damping Factor

To save memory, we used the same message schedule as suggested by Frey. In [FD07a] it is stated that by standard a damping factor $\lambda = 0.5$ has been used. In our experience the standard damping factor λ should be 0.85 since it seems to be a good trade-off between runtime and robustness. For certain datasets it might be wise to reduce λ to tweak the performance, while for other datasets only a larger value as $\lambda = 0.9$ may lead to convergence.



(a) (b)

Fig. 7.5: We have used the toy dataset from Figure 8.2. AP does not converge for all damping factors λ. The algorithm generally converges for large λ. However, as one can see in (a) [for $\lambda \approx 0.85$], the increase in the runtime is exponential: that means, the algorithm is likely to need more than the predefined maximal number of iterations to converge when λ is large. In our experience, damping factors $\lambda \approx 0.8$ usually give good results in satisfactory runtime. To be on the save side, we used $\lambda = 0.9$ throughout this thesis. In this experiment, the AP stops when in 50 consecutive iterations the cluster labels to not change.

7.10.3 The Memory Problem

The algorithm needs memory for the similarities s, responsibilities r and availabilities a for each pair of data points. For 5000 data points saved as 8 byte floats this results in a memory footprint of approximately 572 Mbyte. If the message passing schedule is batch processing, then also a copy of all responsibilities have to be saved. This leads to $4 \times 8 \times 5000^2 \approx 762$ Mbyte. One way to consider more data points is to make the network sparse. This works, if the similarity between two datapoints is $-\infty$. Then one can simply leave out the connecting edge between the two data points.

7.10.4 The Runtime per Iteration

At each iteration, the algorithm has to go over all edges that are not $-\infty$. When the graph is dense, then it needs $\mathcal{O}(N^2)$ arithmetic operations. It neither depends on the number of clusters nor on the dimensionality.

7.11 The Properties of AP's Objective Function

In Section 3.3 we have formulated AP's objective function as K-medoids problem with some kind of minimal description length regularization that influences how many clusters AP finds. Here, we investigate how well AP manages to optimize that objective function. Our focus is on the estimated number of clusters and not on the K-medoids objective function. The results are summarized in Figure 7.6.

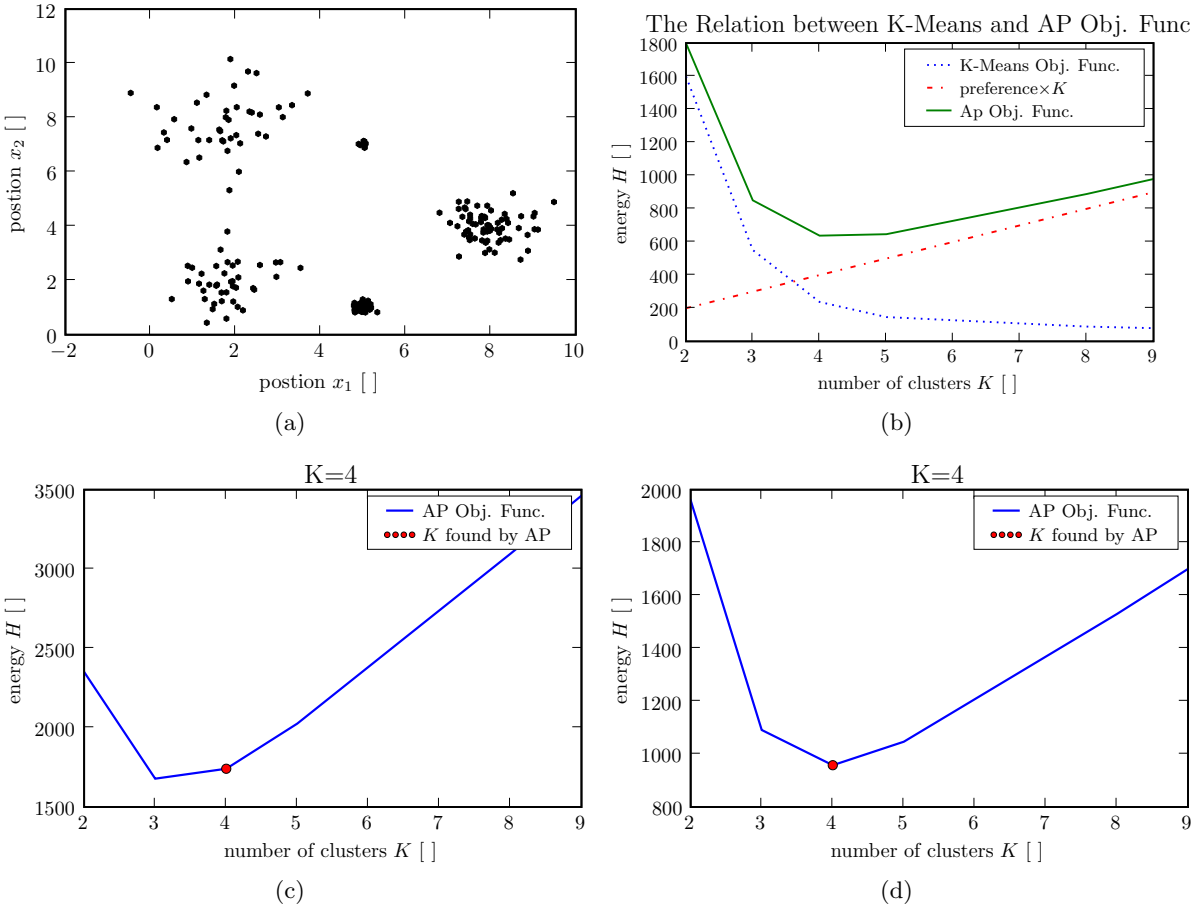


Fig. 7.6: AP's objective function has incorporated self-similarities of exemplars for each data point which are called preferences. By varying the self-preferences one can tune the system to find a specified number of clusters K. In the plots above, the objective function of AP is depicted. The red dot tells us, whether AP has found the best solution. To find the K-means objective function we have used the K-Means algorithm and reran it 400 times to be sure that we have found the best solution. In d) the relation between the K-Means objective function and the AP objective function is depicted. Here, we have used the preference $p = 100$.

8 Cluster Validation (CV)

Data clustering is an unsupervised learning problem. Therefore, there is often no a priori knowledge of how many clusters, if any at all, are in the data. *Cluster validation* (CV) is a technique that tries to evaluate the *right* number of clusters. It belongs to the model selection process. There are several techniques to estimate the number of clusters.

1. Heuristics based on the inner workings of an algorithm (e.g. dendrograms). These methods cannot be used easily on other algorithms.

2. Techniques that operate on the clustering solution and therefore are independent of the algorithm. Examples are internal indices and techniques that operate on external indices. *Internal indices* operate directly on the solution found by an algorithm, i.e. use the same information to assess the quality of the clustering that the clustering algorithm has used to find its solution. The question is if this criterion could have been implemented in the algorithm beforehand. An *external index* is a measure of agreement between partions. Methods that use external indices are CLEST [FD01] and stability based cluster validation (SBCV) [LRBB04].

In the following we will compare the affinity propagation's cluster validation heuristic to CLEST. AP's cluster validation is an incorporated internal index.

8.1 Heuristic Derivation of CLEST and its Relation to SBCV

SBCV is quite similar to the CLEST approach. In the following we will describe CLEST and try to show the similarities to SBCV.

CLEST is hypothesis testing from the Bayesian point of view. The difference between traditional hypothesis testing and model selection is that there is more than one null hypothesis. The *right* model is assumed to be the one that deviates the most from its null hypothesis. That means, that one is not only confronted with the problem when to reject a null hypothesis but one also has to compare *evidences* of alternate hypotheses. The steps that have to be taken in CLEST are:

1. Define a *clustering measure* that corresponds to how *good* the clustering solution is deemed (e.g. by an external index).

2. Define appropriate null hypotheses $\mathrm{H}_0(K)$, where K is the number of clusters (e.g. $K = 1$ with uniform distribution).

3. Define a *test statistic* that compares the null hypotheses $\mathrm{H}_0(K)$ with their alternate hypothesis $\mathrm{H}(K)$ (e.g. absolute distance between the clustering measure of null hypothesis and alternate hypothesis).

4. Collect evidences for each K, i.e., estimate the test statistics. Possibly reject all alternate hypotheses.

5. Pick the solution with the best evidence.

There are many possible ways to define the hypotheses, test statistics and clustering measures. In the following, we give an example of one choice that has been reported to give good results.

8.1.1 Defining a Clustering Measure

The first choice we have to make is the definition of a *clustering measure*. A natural choice is the cluster stability S. The idea is to use the classifier θ that has been trained in the clustering process on the dataset $\{x_n\}_{n=1}^N$ to predict the cluster labels of another dataset $\{\tilde{x}_m\}_{m=1}^M$. The predicted cluster labels $\tilde{c}_m$ are then compared to the cluster labels c_m generated by the clustering algorithm applied on $\{\tilde{x}_m\}_{m=1}^M$. The better the agreement between the labels, the better the

model is deemed. Since clusters may be labeled differently by the predictor and the clustering algorithm, one has to permute the cluster labels to find maximum overlap. Therefore, the *cluster stability measure* is defined as

$$S := \mathbb{E}_{X,\tilde{X}} \min_{\pi \in \mathcal{S}_K} \frac{1}{M} \sum_{m=1}^{M} \mathbb{1}_{\pi[\theta_X(\tilde{X}_m)] \neq c_m} , \tag{8.1}$$

where $X = (X_1, \ldots, X_N)$ and $\tilde{X} = (\tilde{X}_1, \ldots, \tilde{X}_M)$ are iid random variables. The predictor θ_X is trained on the outcomes $x = (x_1, \ldots, x_N)$ of X and applied to the outcomes $\tilde{x}$ of $\tilde{X}$. $\mathbb{1}$ is the indicator function that returns 1 iff the expression is true . $\mathcal{S}_K$ is the symmetric group.

In practice, the distribution of X is not known and only one dataset $\{x_n\}_{n=1}^N$ is available. To estimate the expectation value $\mathbb{E}_{X,\tilde{X}}$ with only one available dataset, one can resample data to generate *new* datasets. This can be done by dividing the original dataset B times into a learning and testing dataset that are disjoint. The *estimated cluster stability measure* is therefore given as

$$\hat{S} := \frac{1}{B} \sum_{b=1}^{B} \min_{\pi \in \mathcal{S}_K} \frac{1}{M} \sum_{m=1}^{M} \mathbb{1}_{\pi[\theta_{x^b}(\tilde{x}_m^b)] \neq c_m^b} , \tag{8.2}$$

where $x^b = (x_1^b, \ldots, x_{N^b}^b)$ are the data points used to train the classifier θ_{x^b}. It has been pointed out in [LRBB04] that there are possibly better choices of the classifier θ than simply using the one trained in the clustering algorithm. For the sake of simplicity we use the same.

In practice it turns out that finding the optimal permutation can be done in $\mathcal{O}(n + k^3)$. However, the algorithm suggested in [Kuh55] is not particularly easy to implement. The problem to find the best permutation can be stated as the problem to permute columns in a *contingency table* $A \in \mathbb{R}^{K \times \tilde{K}}$. Each element $a_{k,\tilde{k}}$ in the matrix A is the number of elements that are labeled as k and $\tilde{k}$. One has to find the permutation of columns s.t. $trace(A)$ is maximal. Note, that $\tilde{K}$ is not necessarily the same as K. The contingency matrix is given as

$a_{\cdot\cdot}$	$a_{\cdot,1}$	$a_{\cdot,2}$	$\ldots$	$a_{\cdot,K}$
$a_{1,\cdot}$	a_{11}	a_{12}	$\ldots$	a_{1K}
$a_{2,\cdot}$	a_{21}	a_{22}	$\ldots$	a_{2K}
$\vdots$	$\vdots$		$\vdots$	
$a_{\tilde{K},\cdot}$	$a_{\tilde{K}1}$	$a_{\tilde{K}2}$	$\ldots$	$a_{\tilde{K}K}$

, (8.3)

where $a_{\cdot,1}$ ($a_{1,\cdot}$) denotes the sum over all rows (columns) of column (row) 1. To circumvent the problem of finding the best permutation, the literature has come up with alternate external indices as for example FM [FM83] or Jaccard [JD88]. In the following we use FM since it has been reported to give better results than the others [FD01]:

$$FM = \frac{Z - M}{2\left[\sum_{\tilde{k}=1}^{\tilde{K}} \binom{a_{\cdot,\tilde{k}}}{2} \sum_{k=1}^{K} \binom{a_{k,\cdot}}{2}\right]^{\frac{1}{2}}} , \tag{8.4}$$

where M is the number of data points in the test dataset and $Z = \sum_{k=1}^{K} \sum_{\tilde{k}=1}^{\tilde{K}} a_{k\tilde{k}}^2$.

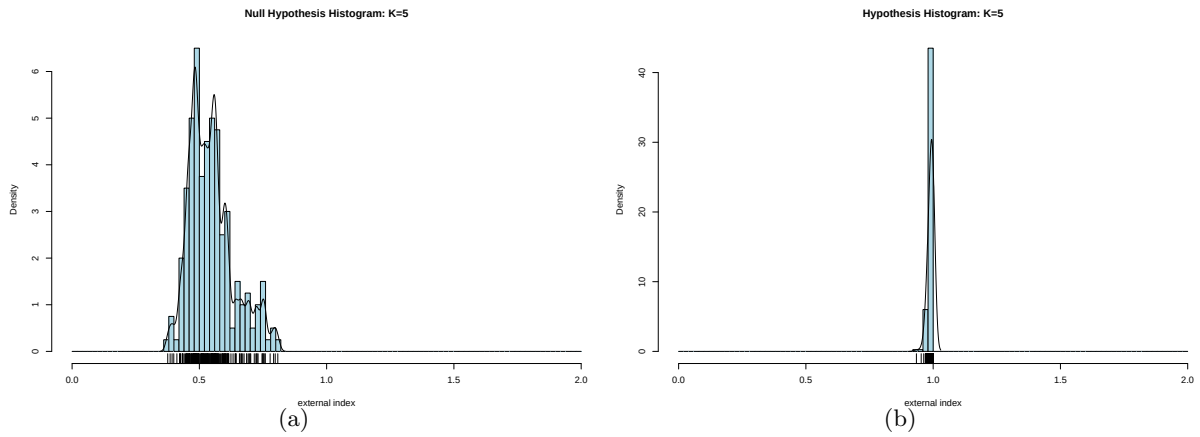
The difference between CLEST and SBCV is that CLEST does not use the B external indices to estimate the cluster stability by an average but to use the median over the B external indices.

8.1.2 Defining an Appropriate Null Hypothesis

We haven't made good experiences with the null hypothesis with uniformly distributed data points (see Figure 8.2 (c) for a short discussion). Instead we use a null hypothesis with one cluster that is normal distributed.

8.1.3 Defining a Test Statistic

The big difference between CLEST and SBCV is the way the stability measures are interpreted. CLEST generates B_0 datasets according to a null hypothesis and computes the the estimated stability measure (as average this time, not as median). The test statistic is the difference between the stability measure of the null hypothesis to the hypothesis. Similarly to hypothesis testing, the hypothesis $\mathrm{H}_0(K)$ is only rejected when the observed stability measure has a P value smaller than a predefined α value. The P value is computed in the following way: CLEST collects B and B_0 values of external indices for the hypothesis and null hypothesis. With those values, it is possible to estimate the probability distribution of the external index for the hypothesis and null hypothesis. One tests how big the probability is that $\bar{e}_K$ could be explained by the null hypothesis. If that probability is too large, i.e., larger than α, the hypothesis is rejected. In Figure 8.1 typical histograms are depicted.



(a) (b)

Fig. 8.1: (a) The histogram of external indices generated from the null hypothesis. The median is approximately at 0.5. (b) the histogram of external indices generated from B bootstrap samples of the original data. There is a clear peak at ≈ 1. If the mean of (b) would significantly overlap with the distribution of (a), the P value would be larger than α value and the hypothesis would be rejected.

8.2 Experimental Comparison Between Affinity Propagation's CV and CLEST

For our experiments we used a small toy problem (Figure 8.2 (a)). The CLEST algorithm is summarized in Algorithm 11. As one can see from Figure 8.2 (b) affinity propagation's cluster validation works quite well in comparison to CLEST (Figure 8.2). However, this is a very simple dataset. In Section 19 one can see that AP's internal CV is often inconclusive.

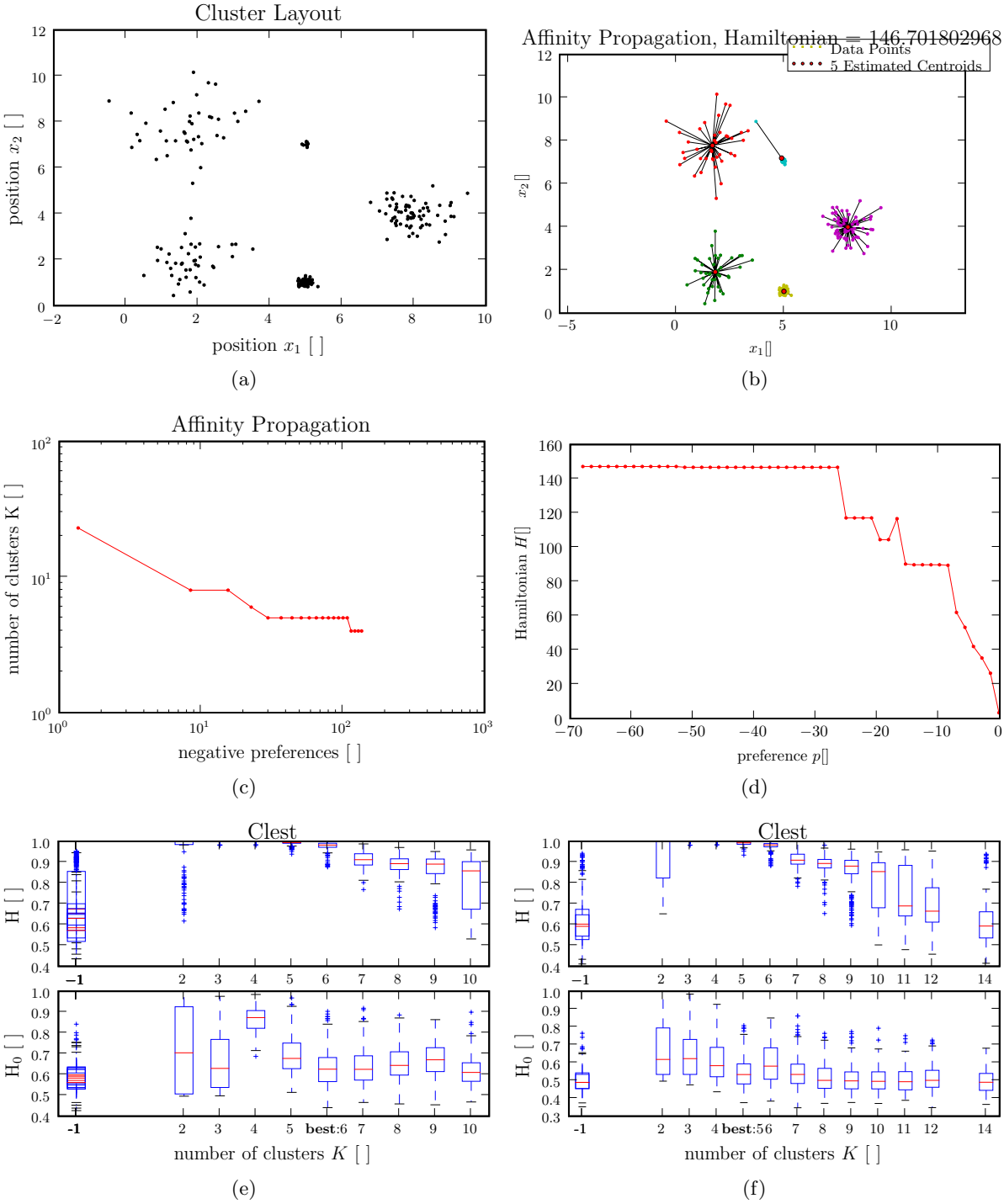


Fig. 8.2: (a) A 2D toy problem with 5 clusters can be seen. The variances and number of datapoints are different for all clusters. (b) Affinity propagation found the *correct* solution with $K = 5$. (c,d) Affinity propagation's internal method to asses the number of clusters. One can see a plateau at 5 clusters. One therefore estimates the number of clusters as 5. (e,f) In the upper subpot one can see boxplots that show how stable a clustering solution is. Especially 3 and 4 clusters are very stable, while 5 has some more variance. 2 is also considered stable but has many outliers. Cluster sizes above 6 do not yield stable solutions are therefore not considered as appropriate model. The hypotheses from the upper subplot have to be compared to the null hypotheses shown in the lower subplot. The distance between the median of the hypotheses and the mean of the null hypotheses is used as evidence for a clustering solution. The cluster number -1 combines all the boxplots of cluster solutions that have been rejected. (e) If the null hypothesis is a uniformly drawn distribution in a rectangle then the 4 cluster solution is quite stable. The reason for that is that the K-means clustering produces compact clusters. 4 balls in a rectangle is apparently more stable than 3 balls. (f) A boxplot with a unimodal normal distribution as null hypothesis. CLEST is able to find the *right* number of clusters, but its a close call since 3 and 4 are also very stable.

foreach *K in K's* **do**
  foreach *b in range(B)* **do**
    Randomly split original dataset $\{x_n\}_{n=1}^N$ into a learning set $\mathcal{L}^b$ and a test set $\mathcal{T}^b$ that are non-overlapping.
    Apply clustering algorithm $\mathcal{A}$ on $\mathcal{L}^b$ to obtain cluster labels c_n.
    Train classifier θ using the learning set $\mathcal{L}^b$.
    Apply clustering algorithm $\mathcal{A}$ on $\mathcal{T}^b$ to obtain cluster labels c_m.
    Use classifier θ on testing set $\mathcal{T}^b$ to obtain new cluster labels $\tilde{c}_m$.
    Compute external index FM and save it as e_{Kb}.
foreach *K in K's* **do**
  foreach b_0 *in range*(B_0) **do**
    Generate dataset $\{x_n\}_{n=1}^N$.
    Randomly split original dataset $\{x_n\}_{n=1}^N$ into a learning set $\mathcal{L}^b$ and a test set $\mathcal{T}^b$ that are non-overlapping.
    Apply clustering algorithm $\mathcal{A}$ on $\mathcal{L}^{b_0}$ to obtain cluster labels c_n.
    Train classifier θ using the learning set $\mathcal{L}^{b_0}$.
    Generate dataset $\mathcal{L}^{b_0}$
    Apply clustering algorithm $\mathcal{A}$ on $\mathcal{T}^{b_0}$ to obtain cluster labels c_m.
    Use classifier θ on testing set $\mathcal{T}^{b_0}$ to obtain new cluster labels $\tilde{c}_m$.
    Compute external index FM and save it as f_{Kb}
Compute average $\bar{e}_K = \frac{1}{B}\sum_{b=1}^B e_{Kb}$.
Compute median $\tilde{f}_K = \text{median}_b f_{Kb}$.
Compute test statistic $d_K = \bar{e}_K - \tilde{f}_K$.
Define $K^- := \{K | p \leq \alpha \text{ and } d_K \geq d_{\min}\}$, where α is the α-value of the null hypothesis and $d_{\min}$ another threshold parameter. The estimated number of clusters is $\hat{K} = \text{argmax}_{K \in K^-} d_K$.

Algorithm 11: The CLEST algorithm. Typical values for the threshold parameters are $d_{\min} = 0.05$ and $\alpha = 0.05$.

9 Mapping Pairwise Clustering Methods to Euclidean Spaces

In this section, we will use matrix notation: $X \in \mathbb{R}^{N \times D}$ is the matrix of data points; row n contains the vector x_n. D is the *distance matrix*. Each element $d_{nm} = \|x_n - x_m\|_2^2$ is the squared Euclidean distance between data point n and m. We use the notion of *kernel matrix* K as the scalar product between data point n and m, i.e., $k_{nm} = \langle x_n, x_m \rangle = x_n^T x_m$.

9.1 From the Kernel Matrix to the Embedding in Euclidean Space

Consider the case, that we are given the data points X. We can therefore compute the kernel matrix $K = XX^T$. With the knowledge of the kernel matrix K, we can compute vectors z_n in a Euclidean vector space that have the same kernel matrix K by a simple eigenvalue decomposition:

$$K = U\Lambda U^T = \underbrace{U\Lambda^{1/2}}_{=:Z}\Lambda^{1/2}U^T = ZZ^T \; . \tag{9.1}$$

since the kernel matrix K is generated from D-dimensional vectors, only D eigenvectors are non-zero. We denoted embedded vectors with z_n and their matrix with Z. This can be seen from the definition of the eigenvectors v_n:

$$XX^T v_n = \lambda_n v_n \; . \tag{9.2}$$

Since X^T is a linear mapping from $\mathbb{R}^N \to R^D$, the rank of XX^T is surely $\leq D$. The following calculation shows how the eigenvectors v_n can be computed

$$\begin{aligned} XX^T v_n &= \lambda_n v_n & (9.3)\\ \Rightarrow X^T XX^T v_n &= \lambda_n X^T v_n & (9.4)\\ \Leftrightarrow X^T X w_n &= \lambda_n w_n \; , & (9.5) \end{aligned}$$

where $X^T X \in \mathbb{R}^{D \times D}$ and $w_n \in \mathbb{R}^{D \times 1}$. This can see in Figure 9.1 (a). Furthermore, it is a good idea to move the average of the data points to the center of the coordinate system, i.e., $x_n^c = x_n - \bar{x}$, where $\bar{x} = \frac{1}{N}\sum_{n=1}^N x_n$ is the estimated expectation value of x. In matrix writing, the centralization reads

$$X^c = X - \frac{1}{N}ee^T X \; , \tag{9.6}$$

where $e = (1, \ldots, 1)^T$.

The *centralized* kernel matrix is therefore given as

$$\begin{aligned} K^c &= X^c (X^c)^T & (9.7)\\ &= \left(X - \frac{1}{N}ee^T X\right)\left(X - \frac{1}{N}ee^T X\right)^T & (9.8)\\ &= \left(\underbrace{1 - \frac{1}{N}ee^T}_{=Q}\right) XX^T \left(1 - \frac{1}{N}ee^T\right) & (9.9)\\ &= QKQ \; , & (9.10) \end{aligned}$$

where $Q = \frac{1}{N}ee^T - \mathbb{1}$.

9.2 From the Kernel Matrix to the Squared Euclidean Distance Matrix

The squared euclidean distance is $d_{nm} = \|x_n - x_m\|_2^2$. It is related to the inner product via

$$d_{nm} = \langle x_n - x_m, x_n - x_m \rangle = \langle x_n, x_n \rangle + \langle x_m, x_m \rangle - 2\langle x_n, x_m \rangle = k_{nn} + k_{mm} - 2k_{nm} \; . \tag{9.11}$$

Written as matrices

$$D = ve^T + ev^T - 2K \; , \tag{9.12}$$

where v is the the diagonal of K.

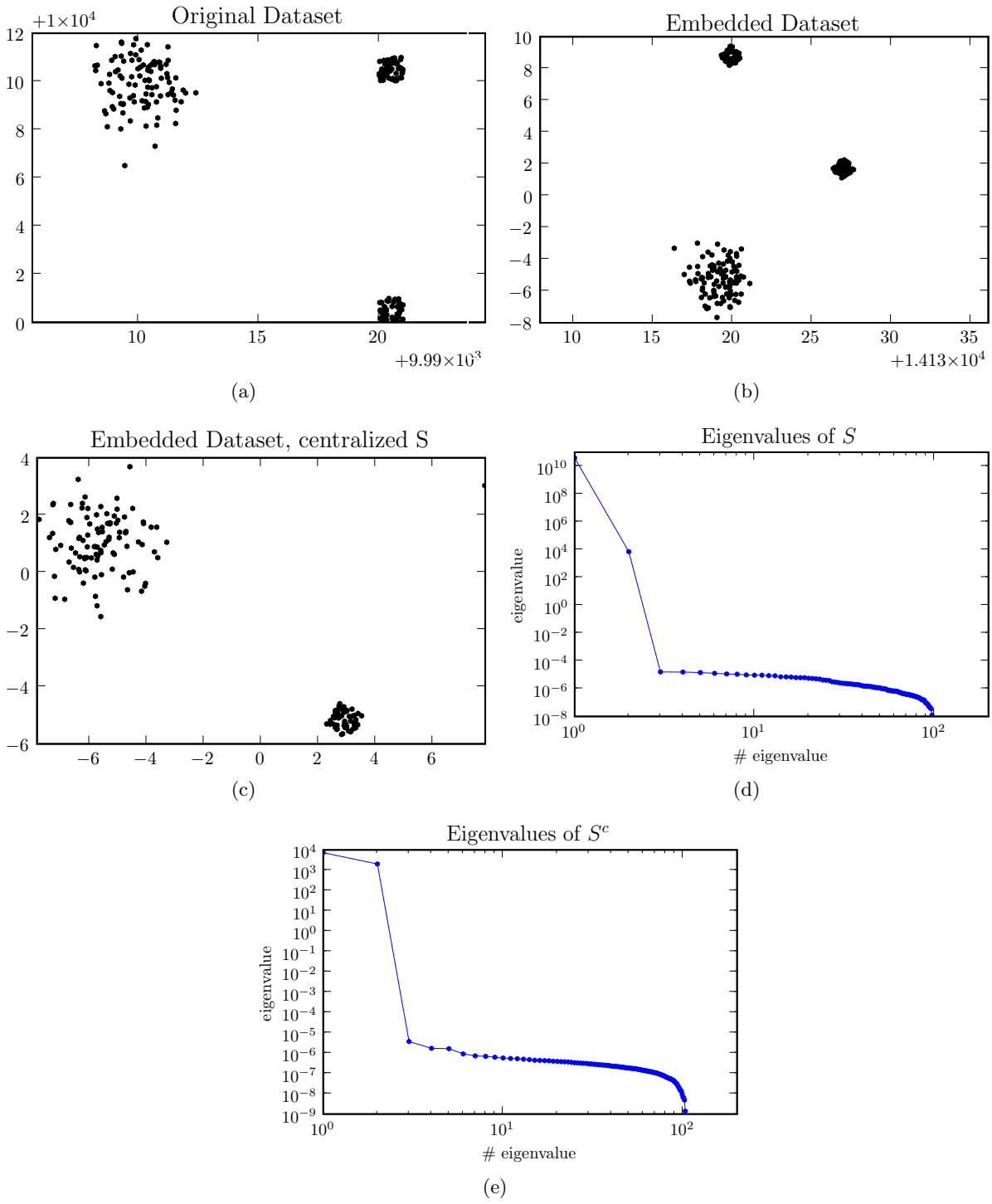


Fig. 9.1: (a) The original dataset. Notice, that the center of mass is far away from the coordinates origin. (b) Shows the embedding computed from $K = XX^T$. In the x-direction there is still a big offset from the origin. (d) shows the embedding of the centralized inner product matrix $K^c = (X - \frac{1}{N}ee^T)(X - \frac{1}{N}ee^T)^T$. One can see that the center of mass of embedded dataset lies in the origin. (d) shows the eigenvalues of the matrix K. The largest eigenvalue is very large and order of magnitudes larger than the second largest eigenvalue. This is inconvenient for numerical reasons. (e) shows the eigenvalues of the centralized inner product matrix K^c. Note, that all eigenvalues but the first two are zero. Their small values can be explained by machine precision problems in the eigenvalue decomposition.

9.3 From the Squared Euclidean Distance Matrix to the Embedding

If not S but D is known, the embedding can still be computed easily because of the following relation between the centralized distance matrix D and S:

$$D^c = -2S^C \ . \tag{9.13}$$

The derivation is linear algebra:

$$\begin{aligned}
QDQ &= Q(ve^T + ev^T)Q - 2K^c \\
&= (1 - \frac{1}{N}ee^T)(ve^T + ev^T)(1 - \frac{1}{N}ee^T) - 2K^c \\
&= \left[(ve^T + ev^T) - \frac{1}{N}ee^Tve^T - \frac{1}{N}e\underbrace{e^Te}_{=N}v^T\right](1 - \frac{1}{N}ee^T) - 2K^c \\
&= ve^T - \frac{1}{N}ee^Tve^T - \frac{1}{N}v\underbrace{e^Te}_{=N}e^T + \frac{1}{N^2}ee^Tv\underbrace{e^Te}_{=N}e^T - KS^c \\
&= -\frac{1}{N}ee^Tve^T + \frac{1}{N}ee^Tve^T - 2K^c \\
&= -2K^c \ .
\end{aligned}$$

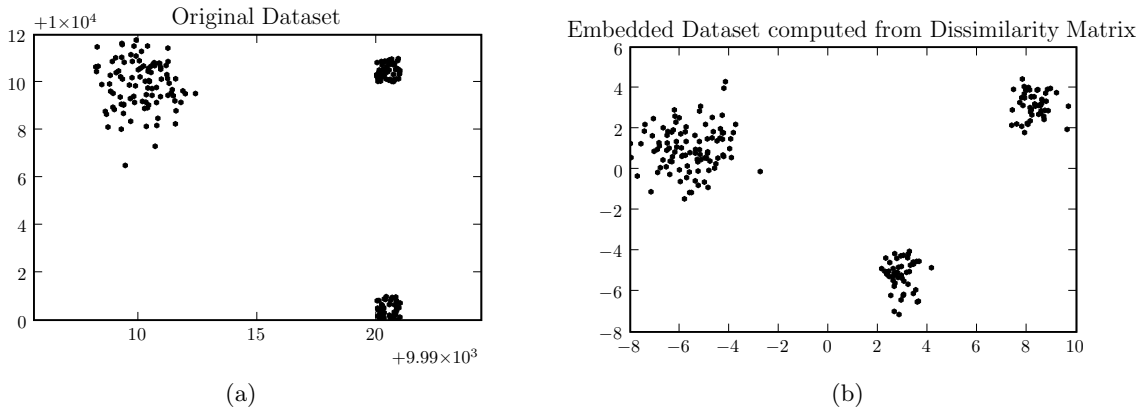


Fig. 9.2: (a) The embedded dataset computed from the perturbed centered distance matrix $\tilde{D}^c$. One can see, that the square-formed clusters are more or less normally distributed when additive Gaussian noise is added to the distance matrix D.

9.4 Constant Shift Embedding

Consider the case, when there is no dataset X in a Euclidean vector space that leads to a given dissimilarity matrix D. This is the case when the kernel matrix K is not positive semi-definite. The only way out is to change the kernel matrix s.t. it becomes positive semi-definite. One way is to substract the same constant to all eigenvalues. The smallest possible value is the smallest eigenvalue $\lambda_{\min}$. We call such a kernel matrix *constant shifted*. Let $K = U\Lambda U^T$ the eigenvalue decomposition. Then $U(\Lambda - \lambda_{\min}\mathbb{1})U^T = K - \lambda_{\min}\mathbb{1}$. The question is how the dissimilarity

matrix D changes, when the kernel matrix is diagonal shifted by a constant:

$$\begin{aligned} D^c &= -2K^c \\ QDQ &= -2QKQ \\ &= -2Q\left(K + (\lambda_{\min} - \lambda_{\min})\mathbb{1}\right)Q \\ &= -2Q\left(K + (\lambda_{\min} - \lambda_{\min})\mathbb{1}\right)Q \\ &= -2Q\left(K - \lambda_{\min}\mathbb{1}Q + \lambda_{\min}\mathbb{1})\right)Q \\ Q\left(D - 2\lambda_{\min}\mathbb{1}\right)Q &= -2Q(K - \lambda_{\min}\mathbb{1})Q \\ QD^{\mathrm{CSE}}Q &= -2QK^{\mathrm{CSE}}Q\ , \end{aligned}$$

where the *constant shift embedded* (CSE) dissimilarity matrix D^{CSE} is given by

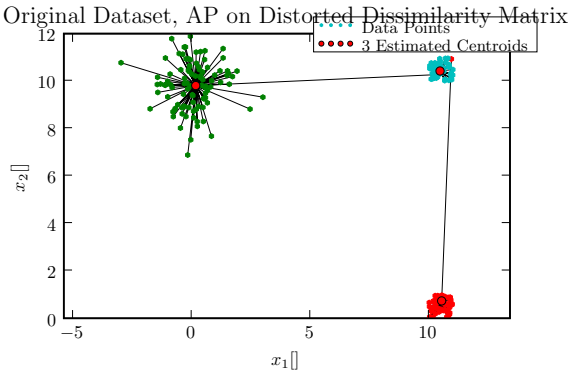
$$D^{\mathrm{CSE}} = D - 2\lambda_{\min}Q\ . \tag{9.14}$$

The pairwise clustering objective function is invariant under such off-diagonal shifts up to a constant:

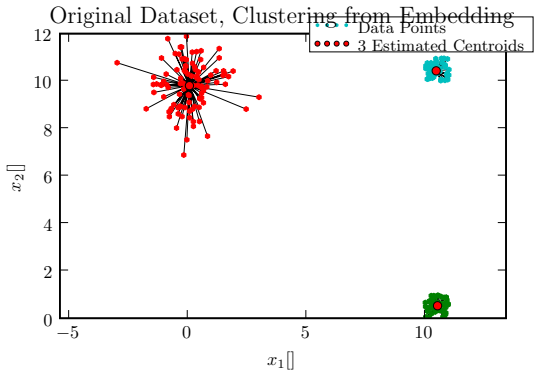
$$\begin{aligned} H^{\mathrm{CSE-PC}}(c) &= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{M_{kn}M_{km}}{|C_k|}d_{nm}^{\mathrm{CSE}} \\ &= \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{M_{kn}M_{km}}{|C_k|}d_{nm} - 2\lambda_{\min}(1-\delta_{nm}) \\ &= -2\lambda_{\min}(N-K) + \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{M_{kn}M_{km}}{|C_k|}d_{nm} \\ &= -2\lambda_{\min}(N-K) + H^{\mathrm{PC}}(c) \end{aligned} \tag{9.15}$$

9.5 Distorting The Distance Matrix

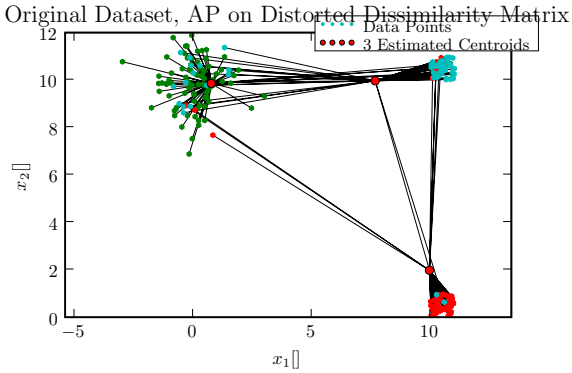
We are given a general distance matrix that is not symmetric and not positive definite. AP can run directly on such a distance matrix. We are interested in how AP performs on such data compared to the K-Means algorithm on an embedded dataset. As dimension of the vector space we chose $D = 2$, since our original dataset, that is used to compute the distance matrix, is also two dimensional. As one can see in Figure 9.3, the denoising that comes with the PCA of S, results in a better clustering solution than directly using AP.



(a) Additive Gaussian noise with variance 50.



(b) Additive Gaussian noise with variance 100.



(c) Additive Gaussian noise with variance 100.

Fig. 9.3: We add additive Gaussian noise on the distance matrix, run AP on that distance matrix, obtain the cluster labels and then plot the clustering solution of the original dataset. In (a) one can see that for a variance of 50, AP gives still a quite reliable clustering. In (b) one can see, that for larger variances, a denoising is necessary to get a good clustering solution.

10 Graph Partitioning by Affinity Propagation

The graph partitioning problem asks to partition a dataset into clusters based on information saved in the weights associated with edges on a graph $(\mathfrak{G}, V, E)$. V is the set of all vertices and E the set of all edges.Since the whole information saved in the weights of the edges, the graph can be represented as *adjacency matrix* A. The weight of an edge from data point n to data point m is denoted a_{nm}. The adjacency is a similarity measure.

10.1 Normalized Cut and the Weighted K-Means Objective Function

We focus on the normalized cut since it is a popular method for image segmentation. The literature uses different notations for the normalized cut problem. Here, we state *normalized cut* objective function in the notation introduced by Shi and Malik [SM00], reformulate it using the notation used by Dhillon et al [DGK07] and finally translate it to a weighted pairwise clustering problem:

$$
\begin{aligned}
H^{\mathrm{Ncut}}(c) &= \min_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{\mathrm{cut}(C_k, V \setminus C_k)}{\mathrm{assoc}(C_k, V)} && (10.1)\\
&= \min_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{\mathrm{links}(C_k, V \setminus C_k)}{\mathrm{degree}(C_k, V)} && (10.2)\\
&= \min_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{\mathrm{links}(C_k, V \setminus C_k)}{\mathrm{links}(C_k, V)} && (10.3)\\
&= \min_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{\mathrm{links}(C_k, V) - \mathrm{links}(C_k, C_k)}{\mathrm{links}(C_k, V)} \\
&= K - \max_{C_1,\ldots,C_K} \sum_{k=1}^{K} \frac{\mathrm{links}(C_k, C_k)}{\mathrm{links}(C_k, V)} \\
&= K - \max_{c} \sum_{k=1}^{K} \frac{\sum_{n\in C_k} \sum_{m\in C_k} a_{nm}}{\sum_{n\in C_k} \sum_{m\in V} a_{nm}} \\
&= K - \max_{c} \sum_{k=1}^{K} \frac{\sum_{n=1}^{N} \sum_{m=1}^{N} M_{kn} M_{km} a_{nm}}{\sum_{n=1}^{N} \sum_{m=1}^{N} M_{kn} a_{nm}} \\
&= K - \max_{c} \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{m=1}^{N} \frac{M_{kn} M_{km}}{\sum_{n=1}^{N} M_{kn} w_n} a_{nm} \\
&= K + \min_{c} \underbrace{\sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{m=1}^{N} \frac{w_n M_{kn} w_m M_{km}}{\sum_{n=1}^{N} M_{kn} w_n} \frac{-a_{nm}}{w_n w_m}}_{=H(c)}, && (10.4)
\end{aligned}
$$

where $\mathrm{cut}(C_k, V \setminus C_k) = \sum_{n\in C_k} \sum_{m\in V\setminus C_k} a_{nm}$ is the sum of edge weights a_{nm} that need to be cut to separate the vertices n in cluster C_k from the rest of the graph. $\mathrm{assoc}(C_k, V) = \sum_{n\in C_k} \sum_{m\in V} a_{nm}$ is the sum of all edge weights that have vertices in C_k as source. degree and links is an equivalent notation. $\mathrm{links}(V, W)$ is simply the sum of all edge weights between vertices in V and W. The weights w are defined as $w_n = \sum_{m=1}^{N} a_{nm}$.

We arrive at the objective function

$$
H(c) = \sum_{k=1}^{K} \sum_{n=1}^{N} \sum_{m=1}^{N} \frac{\tilde{M}_{kn} \tilde{M}_{km}}{|\tilde{C}_k|} (-\tilde{a}_{nm}); , \qquad (10.5)
$$

where $\tilde{a}_{nm} = \frac{a_{nm}}{w_n w_m}$, $\tilde{M}_{kn} = w_n M_{kn}$ and $|\tilde{C}_k| = \sum_{n=1}^{N} \tilde{M}_{kn}$. The weights w are defined as $w_n = \sum_{m=1}^{N} a_{nm}$. $-\tilde{a}_{nm}$ represent a dissimilarity between data points n and m; i.e., if $-\tilde{a}_{nm} > -\tilde{a}_{nl}$

then data point m lies farther away from n than l. However, there is a simple transformation that allows an easier interpretation:

$$
\begin{aligned}
2\min_c H(c) &= \min_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\left[\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}(\underbrace{\tilde{a}_{nn}+\tilde{a}_{mm}-2\tilde{a}_{nm}}_{=:d_{nm}}) - \frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\tilde{a}_{mm} - \frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\tilde{a}_{nn}\right] \\
&= \min_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm} - \max_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\tilde{a}_{mm} - \max_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\tilde{a}_{nn} \\
&= \min_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm} - \max_c \sum_{k=1}^{K}\sum_{m=1}^{N}\tilde{M}_{km}\tilde{a}_{mm} - \max_c \sum_{k=1}^{K}\sum_{n=1}^{N}\tilde{M}_{kn}\tilde{a}_{nn} \\
&= \min_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm} - 2\max_c \sum_{k=1}^{K}\sum_{n=1}^{N}\tilde{M}_{kn}\tilde{a}_{nn} \\
&= \min_c \sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm} - 2\max_c \sum_{n=1}^{N} w_n\tilde{a}_{nn}\underbrace{\sum_{k=1}^{K}M_{kn}}_{=1} \\
&= \min_c \underbrace{\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm}}_{=2H^{\mathrm{WPC}}} \underbrace{-2\sum_{n=1}^{N} w_n\tilde{a}_{nn}}_{=\mathrm{const}}
\end{aligned}
$$

We therefore define the *weighted pairwise clustering* objective function as

$$H^{\mathrm{WPC}}(c) = \frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm}\ , \tag{10.6}$$

where $d_{nm} := \frac{a_{nn}}{w_n^2} + \frac{a_{mm}}{w_m^2} - 2\frac{a_{nm}}{w_n w_m}$. The kernel matrix to that dissimilarity is A. If A is positive semi-definite, then there exist vectors x_n in an $N-1$ dimension vector space such that their distances are $\|x_n - x_m\|_2^2 = d_{nm}$.

Then, the weighted pairwise clustering objective function is equivalent to the *weighted K-means* (WKM) objective function

$$H^{\mathrm{WKM}}(c) = \sum_{k=1}^{K}\sum_{n=1}^{N}\tilde{M}_{kn}\|\tilde{y}_k - x_n\|_2^2\ , \tag{10.7}$$

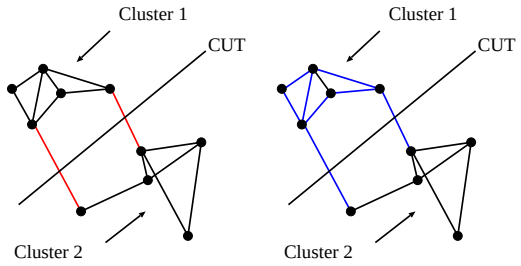
where the minimization of $\tilde{y}_k$ at fixed configuration is computed as

$$\tilde{y}_k = \frac{1}{|\tilde{C}_k|}\sum_{n=1}^{N}\tilde{M}_{kn}x_n\ . \tag{10.8}$$

The proof is exactly the same as in Section 3.3: one has only to replace M by $\tilde{M}$, C by $\tilde{C}$ and y by $\tilde{y}$.

10.2 The Shift-Invariance of the Weighted Pairwise Clustering Objective Function

We know that the PC objective function is invariant under a constant off-diagonal shift up to a constant. Here, we want to show, under what shift the weighted PC objective function is



(a) Cut. (b) Assoc.

Fig. 10.1: (a) The cut sums up the red edges, i.e., the edges that have to be cut to separate cluster 1. (b) The association of cluster 1 is the sum of all blue edges.

invariant such that the dissimilarities can be derived as squared Euclidean distances.

$$\begin{aligned}
H^{\mathrm{CS-PC}}(c) &= \frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm}^{\mathrm{CS}} \\
&= \frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\left(d_{nm}+\lambda\frac{1}{w_n}(1-\delta_{nm})\right) \\
&= \frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}d_{nm}+\frac{\lambda}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\frac{1}{w_n}-\frac{\lambda}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}\frac{\delta_{nm}}{w_n} \\
&= H^{\mathrm{WPC}}+\frac{\lambda}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{M_{kn}\tilde{M}_{km}}{|\tilde{C}_k|}-\frac{\lambda}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\frac{\tilde{M}_{kn}M_{km}w_n^2}{|\tilde{C}_k|}\frac{\delta_{nm}}{w_n} \\
&= H^{\mathrm{WPC}}+\frac{\lambda}{2}(N-K)\,. \qquad (10.9)
\end{aligned}$$

In matrix notation: $D^{\mathrm{CS}} = D + \lambda W^{-1}Q$. Let K be the kernel matrix to $D = (d_{nm})$. We need to know how K can be transformed to make all eigenvalues positive under the constraint s.t. the WPC objective function stays invariant up to a constant:

$$\begin{aligned}
D^{\mathrm{c}} &= QDQ \\
Q\left(D-\lambda W^{-1}Q+\lambda W^{-1}\right)Q &= -2K^{\mathrm{c}} \\
QD^{\mathrm{CS}}Q &= -2Q\left(K-\frac{\lambda}{2}W^{-1}\right)Q\,. \qquad (10.10)
\end{aligned}$$

That means, one has to find a λ s.t. the eigenvalues of $K - \frac{\lambda}{2}W^{-1}$ are non-negative.

10.3 Weighted K-Medoids and Weighted AP

In the weighted K-medoids problem, we search for K exemplars such that the weighted sum of distances $\tilde{a}_{ne_k}$ from the n'th data point to the k'th exemplar is minimal. Formally, the *weighted K-medoids* objective function is given by

$$\begin{aligned}
H^{\mathrm{K-Medoids}}(e) &= \sum_{m=1}^{N}\sum_{n=1}^{N}\tilde{M}_{mn}d_{nm} \\
&= \sum_{m=1}^{N}\sum_{n=1}^{N}M_{mn}w_n d_{nm} \\
&= \sum_{m=1}^{N}\sum_{n=1}^{N}M_{mn}\tilde{d}_{nm}\,, \qquad (10.11)
\end{aligned}$$

where we have defined $\tilde{d}_{nm} := w_n d_{nm}$. M_{mn} is one iff n has picked m as exemplar. The weights modify the dissimilarity matrix. Therefore, the weighted K-means problem can be formulated as standard K-medoids problem. The relation to AP is therefore directly given by Eqn. (3.16). Figure 10.2 shows a solution found by AP when the similarity matrix is weighted according to $\tilde{s} = Ws$.

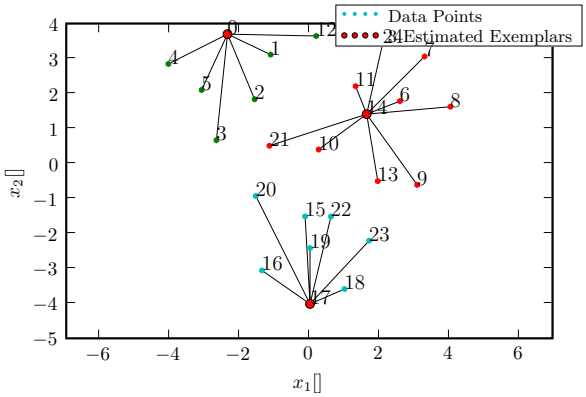
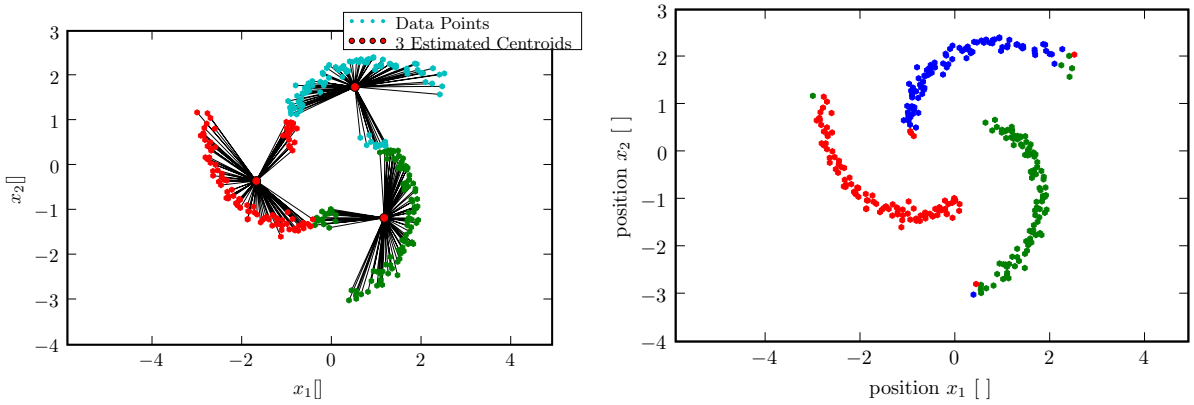


Fig. 10.2: A weighted K-medoids solution found by weighed AP. All weights are set to one. Only w_0 and w_{17} are set to 100.

Translated to the ncut problem, the weighted dissimilarities are $\tilde{d}_{nm} = w_n(\frac{a_{nn}}{w_n^2} + \frac{a_{mm}}{w_m^2} - 2\frac{a_{nm}}{w_n w_m})$. The similarity matrix is

$$s_{nm} = -\tilde{d} = -w_n(\frac{a_{nn}}{w_n^2} + \frac{a_{mm}}{w_m^2} - 2\frac{a_{nm}}{w_n w_m}) \,. \tag{10.12}$$

Written as matrices $S = W^{-1}ve^T + Wev^TW^{-2} - 2AW^{-1}$, where $W = \mathrm{diag}(w_1, \ldots, w_N)$, $v = (a_{11}, \ldots, a_{NN})^T$ and $w_m = \sum_{n=1}^{N} a_{mn}$.

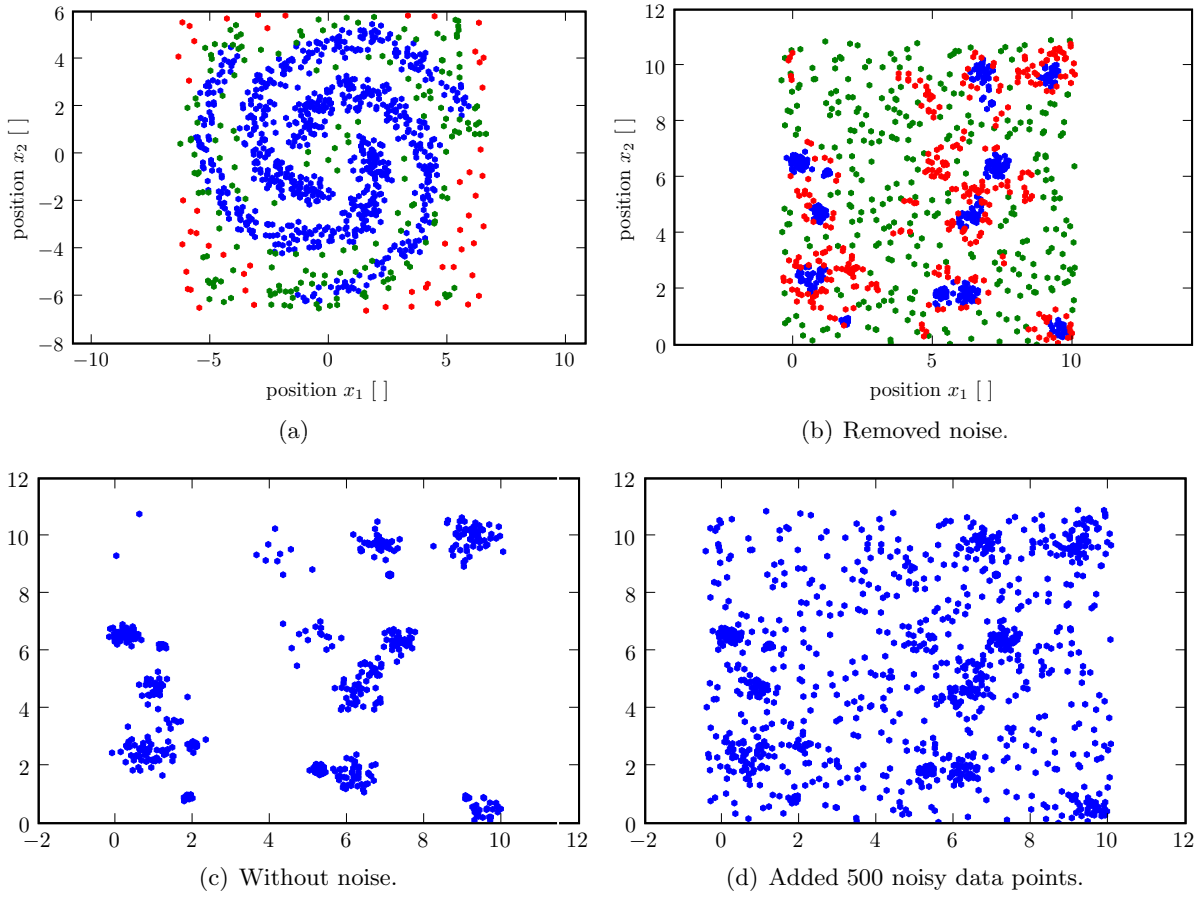


(a) K-Means solution. (b) N-Cut solution.

Fig. 10.3: (a) A typical K-means clustering solution on elongated clusters. The clusters found by K-means are compact and are not an appropriate model order for that dataset. (b) The Ncut clustering solution found with a linear kernel. Using AP on the ncut objective function classifies some of the outer datapoints in a rather unintuitive way.

10.4 Additional Tests

We tried more difficult toy datasets as depicted in Figure 10.4 (a). Currently, we cannot explain the observed clustering solution. It looks as if the algorithm separates regions by their density. To test our hypothesis, we generated a dataset and added uniform noise (c.f. Figure 10.4). The algorithm does in fact separate the clusters from the noisy background.



(a)

(b) Removed noise.

(c) Without noise.

(d) Added 500 noisy data points.

Fig. 10.4: (a) The spiral arms are not clustered. Rather, the algorithm separates regions by their density. (b-d) Using AP with the similarity matrix for Ncut (RBF kernel) clusters the data by their density.

11 Comparision of AP and K-Means Algorithm for Kernel K-Means Clustering

The idea of kernel K-means is to map the data points to a high-dimensional space. We denote the mapping by $\phi(x_n)$. The matrix of all mapped data points is therefore $\phi(X)$. The *kernel matrix* K is defined as

$$K = \phi(X)\phi(X)^T \ , \tag{11.1}$$

where $X \in \mathbb{R}^{N \times D}$ the matrix of datapoints. For any K that is positive semi-definite one can derive vectors z_n such that their inner product is $ZZ^T = K$.

11.1 The Radial Basis Function Kernel

A popular kernel matrix is the *radial basis function*kernel (RBF). It is defined as

$$K_{nm} = e^{-\frac{\|x_n - x_m\|_2^2}{2\sigma^2}} \ . \tag{11.2}$$

σ is a control parameter that can be tuned. The embedding can be computed as $Z = U\Lambda^{1/2}$, where $K = U\Lambda U^T$ is the eigenvalue decomposition.

11.2 Experimental Comparison

On the dataset Z one can run the K-means algorithm and one can also easily compute a similarity matrix for AP as negative squared Euclidean distances $s_{nm} = -\|z_n - z_m\|_2^2$ between two data points z_n and z_m. In Figure 11.1 one can see that AP has problems to separate the three spiral arms.

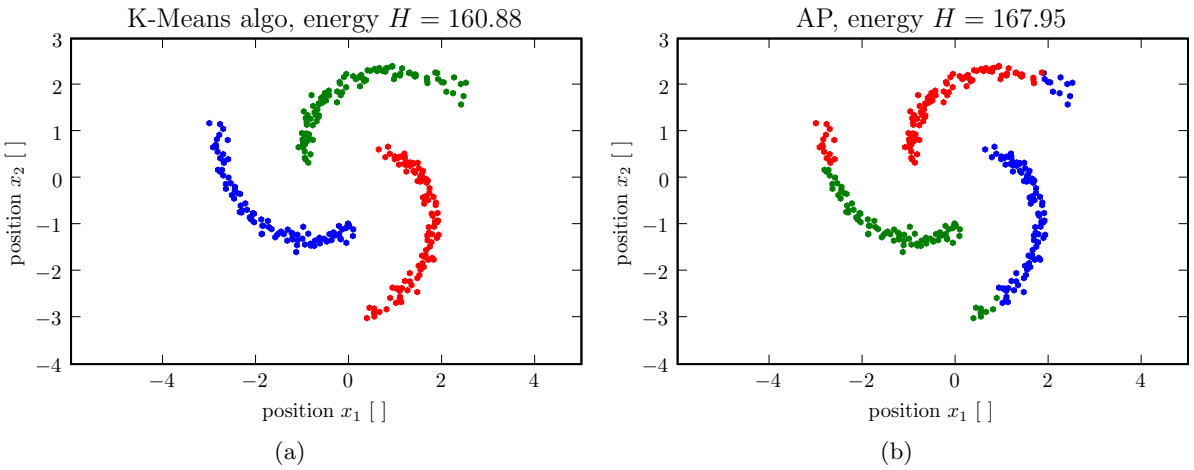


Fig. 11.1: (a) Solution found by the K-means algorithm. (b) Solution found by AP. AP failed to separate the spiral arms.

12 Influence of Noisy Dimensions on Affinity Propagation

Since AP failed to find the right clustering solution in a high-dimensional space we test how noisy dimensions affect the performance of the K-means algorithm and AP. As one can see in Figure 12.1, the K-means algorithm gives better results. The setup is the following: we create two clusters in two dimensions with fixed variance var=$\sigma^2 = 1$. The dimension is fixed at $D = 100$. In the remaining 98 dimensions we add Gaussian noise with increasing variance. The *risk* R is the expected loss. It is defined as $R = \frac{1}{N}\sum_{n=1}^{N}[c_n^{\text{gt}} \neq c_n]$, where c_n^{gt} are the ground-truth labels.

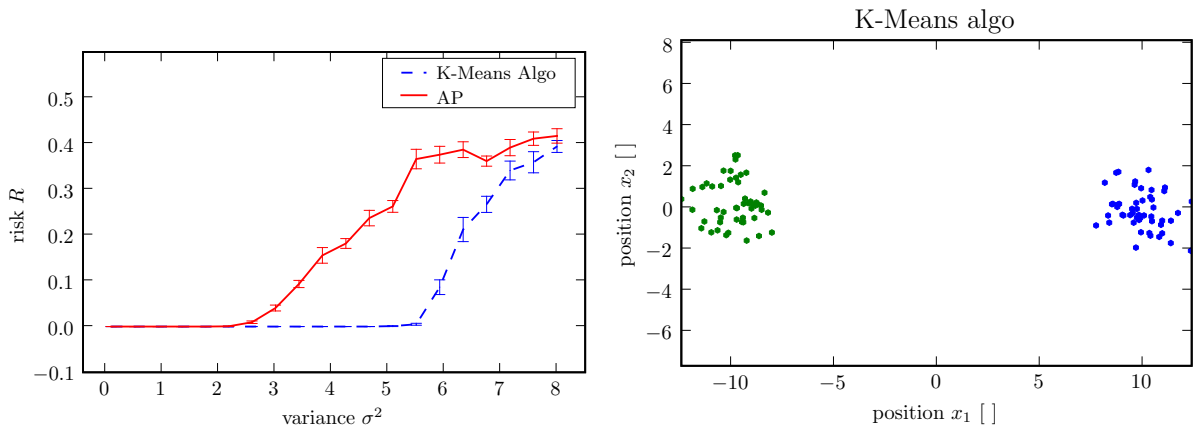


Fig. 12.1: (a) When the variance of the Gaussian noise in the 98 dimensions is increased, AP soon fails to find the right clustering solution and is outperformed by the K-means algorithm. We plotted the mean of 20 runs together with its standard error. (b) The first two dimensions of a typical cluster layout. The variance of the first two dimensions is one for each cluster. In the remaining 98 dimensions the variance can be much higher.

13 Path-based Clustering by Affinity Propagation

When the clusters in the data are not compact, but elongated, the K-means objective function is not the best choice. Using K-means leads to a *wrong model order* (c.f. Figure 13.1)

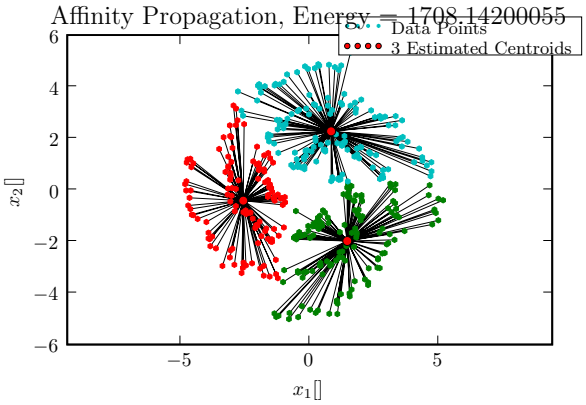


Fig. 13.1: A wrong model-order. The clustering solution is computed by use of the K-means objective function on the spiral arms dataset. The spiral arms cannot be identified.

Path- based clustering (PBC) propose uses the pairwise clustering objective function

$$H^{\mathrm{PBC}}(c) = \sum_{k=1}^{K} \frac{1}{|C_k|} \sum_{n=1}^{N} \sum_{m=1}^{N} d_{nm}^{\mathrm{PBC}} \ , \tag{13.1}$$

where the path-based clustering dissimilarities d are defined as

$$d_{nm}^{\mathrm{PBC}} = \min_{\pi \in \mathcal{P}_{nm}} \left[\max_{1 \le o \le |\pi|} d_{\pi_o \pi_{o+1}} \right] \ , \tag{13.2}$$

where $\mathcal{P}_{nm}$ is the set of all possible paths from data point n to data point m. π_o is the o'th vertex on the path from n to m. In words: When a path has been chosen, then the dissimilarity is the largest distance between two data points on the path. Try every path and take the one which gives the smallest value.

The path-based clustering objective function is the same as the pairwise clustering function and therefore we can find an embedding in an $N-1$ dimensional space, solve there the K-means problem and we have therefore solved the path-based clustering problem. This works, since d leads to a positive semi-definite matrix. Affinity propagation is well-suited to solve this high-dimensional problem since it operates directly on a similarity matrix. This is a big advantage over iterative algorithms as the K-means algorithms that scale with $\mathcal{O}(D)$ since $D = N-1$ grows with N. A principal component analysis of the high-dimensional space may destroy important information about the structure in the data.

Explicitly, once d^{PBC} has been computed, the similarities are

$$s_{nm} = -d_{nm}^{\mathrm{PBC}} \ . \tag{13.3}$$

This similarity matrix can be fed to AP without further pre-processing. First tests showed, that solves the optimization to satisfactory extent. AP is very sensitive to noise on the similarity matrix. Since many entries in the similarity matrix are the same, we wondered if that might lead to a degenerate state and added a normally distributed noise $\sim \mathcal{N}(0, 10^{-10})$ to the similarity matrix. Surprisingly, AP now finds a quite different solution as one can see in Figure 13.2.

13.1 Computation of the Path-Based Dissimilarities

AP needs $\mathcal{O}(N^2)$ operations per iteration. For path-clustering to work efficiently as algorithm, it is necessary that the dissimilarities d^{PBC} can be computed in low runtime. In [Fis06] it

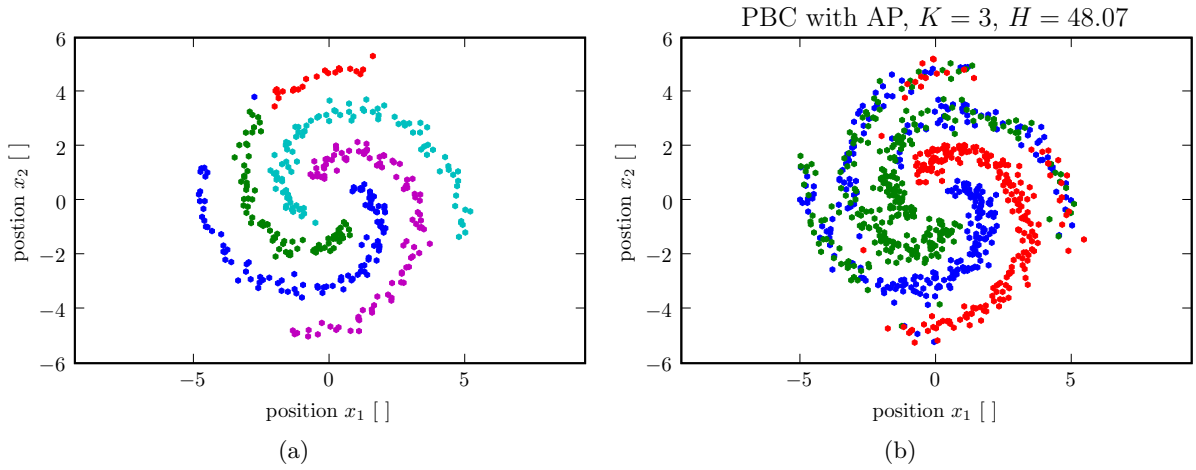


Fig. 13.2: Two toy datasets with $N = 400$ (a) and $N = 800$ (b) data points. In (a) path-based clustering finds a good solution. In (b) we added a tiny amount of normally distributed noise [$\sim \mathcal{N}(0, 10^{-10})$]on the similarity matrix. Compared to Figure 13.3 this solution is much worse.

is proved that the runtime scales as $\mathcal{O}(N^2 \log N)$. In practice, one is restricted to less than 4000 data points due to the $\mathcal{O}(N^2)$ scaling and the memory problem. In that regime, $\log N$ is much smaller than the average number of iterations needed by AP. Therefore, the time for the preprocessing is of minor importance in practice.

13.2 Comparison: PBC with AP vs K-Means Algorithm

Here we show a performance comparision between the solution found by AP and by the K-means algorithm. As one can see in Figure 13.3, the K-means algorithm (with 40 restarts) finds a solution with lower energy H. For both AP and K-means algorithm, we computed the energy in the following way: From the path-based distances d_{nm}^{PBC} we computed embedded vectors z_n in N dimensions. We run AP on $s_{nm}^{\mathrm{PBC}} = -d_{nm}^{\mathrm{PBC}}$ and K-means on the dataset z_n. With the returned cluster labels, we computed centroids in the N dimensional space and computed the K-means objective function.

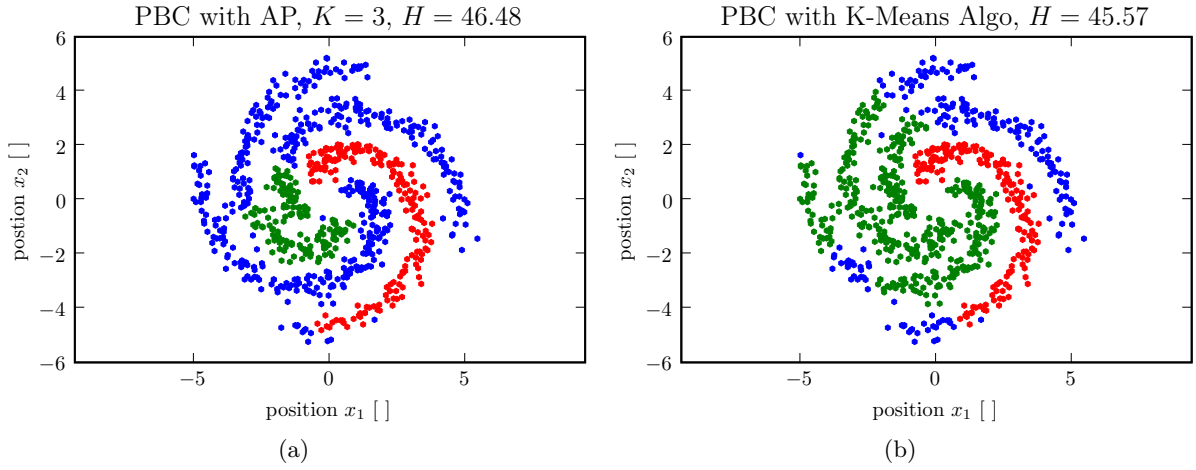


Fig. 13.3: (a) The solution found by AP. AP's solution is not as good as the K-means algorithm's. The fact that K-means scatters parts of the blue cluster is most likely a deficiency of the objective function and not a poor solution in the optimization.

14 Possible Variations of Affinity Propagation

As described in the previous sections, the approach of message passing seems to be a good one. There are many similar problems to the optimization of the K-Means objective function. This section assembles some attempts and ideas to generalize message passing algorithms to other objective functions. However, as we will see, most objective functions have a structure that makes impossible, or at least very hard, to implement efficiently as algorithm.

14.1 Necessary Properties of the Objective Function

For an efficient algorithm, the objective function must fulfill the condition that the runtime must not grow faster than $\mathcal{O}(N^2)$. AP needs $\mathcal{O}(N^2)$ and therefore an algorithm with $\mathcal{O}(N^3)$ may need 2000 times as long, when the number of data points is $N = 2000$.

Therefore, variations of AP must meet the following conditions:

1. A necessary condition is that the number of edges in the graph is at most $\mathcal{O}(N^2)$.
2. The messages that are sent along each edge are vector of all possible values a variable can take. E.g., for $\mu_{f\to x}(\mathbf{c})$, the message may consist of $c \in [1, \ldots, N]$. There must be the possibility to reduce the number of values that have to be sent.

14.2 Affinity Propagation with a Fixed Number of Clusters

As proposed by Frey in [FD05] one can add an additional constraint to ensure that the algorithm exactly K clusters as output.

$$H(e) = -\sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{m=1}^{N} \delta_m(e) + g(e)\ , \tag{14.1}$$

$$\delta_m(e) = \begin{cases} \infty & [e_m \neq m] \text{ and } [e_l = m \text{ for a } l] \\ 0 & \text{else} \end{cases} \tag{14.2}$$

$$g(e) = \begin{cases} 0 & \text{if } [K = \sum_{n=1}^{N} [e_n = n]] = 1 \\ \infty & \text{else} \end{cases} \tag{14.3}$$

$$\tag{14.4}$$

where we have used Iverson's notation, i.e., [true] = 1 and [false] = 0.

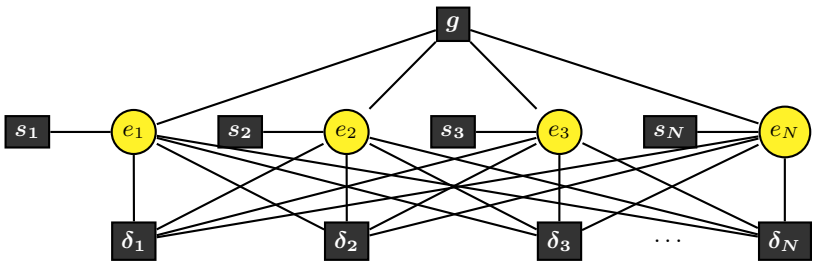


Fig. 14.1: The AP factor graph with an additional node to ensure K clusters.

14.3 An Alternate Factor Graph for the Derivation of an Alternate Affinity Propagation

The derivation of AP is is mostly straight forward. However, the expressions take very long forms that are confusing and provoke mistakes because of typos. Here, we suggest an alternate

factor graph that should have the same result. Possibly, it allows an easier derivation of AP. The corresponding factor graph is depicted in Figure 14.2. The objective function is given as:

$$H(e) = -\sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{m=1}^{N} \delta_{nm}(e_n, e_m) \,, \tag{14.5}$$

$$\delta_{nm}(e_n, e_m) := \begin{cases} \infty & [e_m \neq m] \text{ and } [e_n = m] \\ 0 & \text{else} \end{cases} \,. \tag{14.6}$$

The advantage of this formulation is that it possibly simplifies the case distinction: the message from factor node δ_{nm} to variable node e_n reads:

$$\begin{aligned} \mu_{\delta_{nm}\to e_n}(\mathbf{e}_n) &= \min_{e_m} \delta_{nm}(\mathbf{e}_n, e_m) + \mu_{e_m\to\delta_{nm}}(\mathbf{e}_m) \\ &= \begin{cases} \min_{e_m} \mu_{e_m\to\delta_{nm}}(e_m) & \text{if } e_n \neq m \\ \mu_{e_m\to\delta_{nm}}(m) & \text{if } e_n = m \end{cases} \,. \end{aligned}$$

I.e., there are only two kinds of messages: one for $e_n = m$ and one for $e_n \neq m$. Furthermore, n and m are configuration independent, i.e., when n and m are known, the possible messages are already largely determined.

The disadvantage is that the message $\mu_{\delta_{nm}\to e_m}(\mathbf{e}_m)$ has another structure, i.e.,

$$\begin{aligned} \mu_{\delta_{nm}\to e_m}(\mathbf{e}_m) &= \min_{e_n} \delta_{nm}(\mathbf{e}_n, e_m) + \mu_{e_n\to\delta_{nm}}(\mathbf{e}_n) \\ &= \begin{cases} \min_{e_n} \mu_{e_n\to\delta_{nm}}(e_n) & \text{if } e_m = m \\ \min_{e_m\neq m} \mu_{e_m\to\delta_{nm}}(m) & \text{if } e_m = m \end{cases} \,. \end{aligned}$$

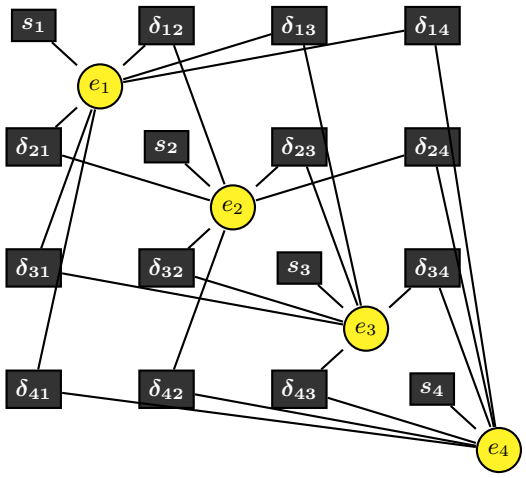


Fig. 14.2: The alternate AP factor graph. It has twice the number of edges than the original AP factor graph.

14.4 Yet Another Alternate Factor Graph for the Derivation of an Alternate Affinity Propagation

Here we present yet another reformulation of the original function that leads to another factor graph (Figure 14.3). This time, there are less edges in the graph. Furthermore, the constraint function is now symmetric in n and m.

$$H(e) = -\sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{m=1}^{N} \delta_{nm}(e_n, e_m) \,, \tag{14.7}$$

$$\delta_{nm}(e_n, e_m) := \begin{cases} \infty & [[e_m \neq m] \text{ and } [e_n = m]] \text{ or } [[e_n \neq n] \text{ and } [e_m = n]] \\ 0 & \text{else} \end{cases} \,. \tag{14.8}$$

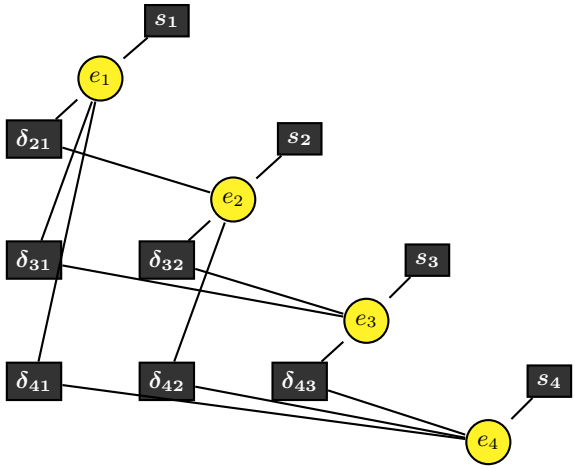


Fig. 14.3: The factor graph for yet another AP objective function (14.5). In this context one should note that the graph has fewer edges than the original AP factor graph which could possibly improve the algorithm. However, first attempts to derive the AP algorithm on that factor graph failed.

14.5 Pairwise Clustering

In this subsection, we investigate whether it is possible to use the sum-product algorithm to solve the optimization problem of pairwise clustering. The pairwise clustering function reads:

$$H^{\mathrm{PW}}(c) = -\frac{1}{2}\sum_{k=1}^{K}\sum_{n=1}^{N}\sum_{m=1}^{N}\underbrace{s(x_m, x_n)\frac{M_{kn}M_{km}}{|C_k|}}_{\equiv f_{knm}(c)} \; . \tag{14.9}$$

The factor graph is depicted in Figure 14.4. The number of edges in the graph is $\mathcal{O}(KN^3)$ and therefore one has not even the chance to derive an efficient implementation by using the tricks used in AP's derivation.

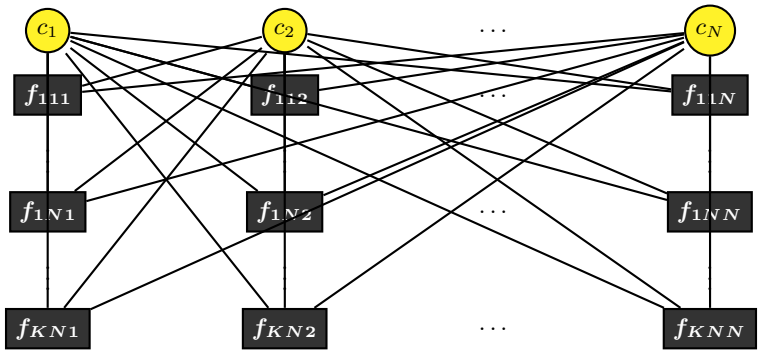
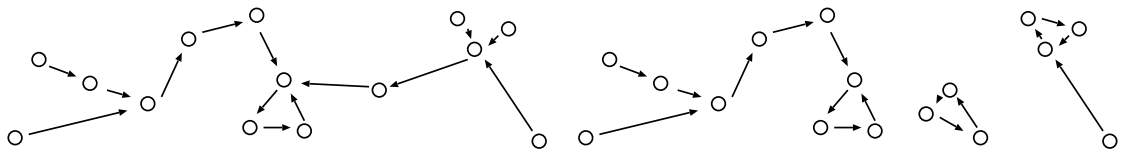


Fig. 14.4: The factor graph for pairwise clustering. Each variable node c_n has N^2K outgoing edges. Therefore, there are N^3K edges in the graph.

14.6 Path Clustering by Relaxing the Constraint of Valid Configurations

If there are no constraint functions that ensure valid configurations at all, the system will find a clustering solution where nearest neighbors pick each other as exemplar. Thus, there will be many clusters with two data points. When there is a constraint which prevents this behavior, a data point may pick any other datapoint as exemplar but this datapoints has to pick it's second nearest neighbor. In the case that this second nearest neighbor's second nearest neighbor is not

the first data point, the datapoints build a chain of exemplars. It may look as in Figure 14.5. However, there is also the possibility that three datapoints build a cluster. I.e., actually, one has to prevent loops in the chain. However, finding a constraint function that allows an efficient implementation is hard.



(a) (b)

Fig. 14.5: How the objective function is supposed to cluster data. In a) the clustering algorithm should find a path through the whole elongated cluster. However, if three of them are placed nearby, they are likely to build a small cluster with three data points. To prevent this behavior one could try resampling. This would eventually remove datapoints that allowed to build small clusters of three data points.

Though the approach is unlikely to work, it shows how the tricks used in the derivation of AP can be applied on other objective functions. The objective function is defined as:

$$H(e) = \sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{n=1}^{N}\sum_{m=1}^{N} \delta_{nm}(e_n, e_m) \tag{14.10}$$

$$\delta_{nm}(e_n, e_m) = \begin{cases} \infty & \text{if } [e_n=m] \text{ and } [e_m=n] \\ 0 & \text{else} \end{cases} . \tag{14.11}$$

In the following we use the property $\delta_{nm}(e_n, e_m) \equiv \delta_{mn}(e_m, e_n)$. In fact, the above formulation uses more edges than necessary. One could save half of it by defining $H(e) = \sum_{n=1}^{N} s(x_{e_n}, x_n) + \sum_{n>m} \delta_{nm}(e_n, e_m)$. We will implicitly us this fact. We hope to find clustering solutions as depicted in Figure 14.5.

The messages that are sent between the nodes are

1. $\mu_{s_n \to e_n}(e_n) = s_n(e_n)$
2. $\mu_{e_n \to s_n}(e_n) = \sum_{m=1}^{N} \mu_{\delta_{nm} \to s_n}(e_n)$
   never used in the update
3. $\mu_{e_n \to \delta_{nm}}(e_n) = s_n(e_n) + \sum_{l \neq \{n,m\}} \mu_{\delta_{nl} \to e_n}(e_n)$
4. $\mu_{\delta_{nm} \to e_n}(e_n) = \min_{\sim e_n} [\delta_{nm}(e_n, e_m) + \mu_{e_m \to \delta_{nm}}(e_m)] = \min_{e_m} [\delta_{nm}(e_n, e_m) + \mu_{e_m \to \delta_{nm}}(e_m)]$

Intermediate Equations: Variable to Factor Analogously to the derivation of AP we start with the case distinction of $\mu_{\delta_{nm} \to e_n}(e_n)$.

$$\begin{aligned} \text{case } e_n = m: \quad \mu_{\delta_{nm} \to e_n}(m) &= \min_{e_m \neq n} [\mu_{e_m \to \delta_{nm}}(e_m)] \\ \text{case } e_n = n: \quad \mu_{\delta_{nm} \to e_n}(n) &= \min_{e_m} [\mu_{e_m \to \delta_{nm}}(e_m)] \end{aligned}$$

For any $l \neq m$ the messages are identical, i.e., $\mu_{\delta_{nm} \to e_n}(n) \equiv \mu_{\delta_{nm} \to e_n}(l)\ \forall l \neq m$.

Again, we define $\mu_{\delta_{nm} \to e_n}(e_n) = \tilde{\mu}_{\delta_{nm} \to e_n}(e_n) + \bar{\mu}_{\delta_{nm} \to e_n}$ with $\bar{\mu}_{\delta_{nm} \to e_n} := \min_{e_m \neq n} [\mu_{e_m \to \delta_{nm}}(e_m)]$ to obtain

$$\begin{aligned} \text{case } e_n = m: \quad \mu_{\delta_{nm} \to e_n}(m) &= \bar{\mu}_{\delta_{nm} \to e_n} \\ \text{case } e_n = n: \quad \mu_{\delta_{nm} \to e_n}(n) &= \min [0, \tilde{\mu}_{e_m \to \delta_{nm}}(n)] + \bar{\mu}_{\delta_{nm} \to e_n} \end{aligned}$$

Intermediate Equations: Factor to Variable The same approach for the messages from the variable nodes e_n to the factor nodes δ_{nm}:
At first, we investigate the case $e_n = m$. We have to make a change of coordinates s.t. these messages are zero.

$$\begin{aligned} 0 \stackrel{!}{=} \tilde{\mu}_{e_n \to \delta_{nm}}(m) &= \mu_{e_n \to \delta_{nm}}(m) + \bar{\mu}_{e_n \to \delta_{nm}} \\ &= s_n(m) + \sum_{l \neq \{n,m\}} \mu_{\delta_{nl} \to e_n}(m) + \bar{\mu}_{e_n \to \delta_{nm}} \\ &= s_n(m) + \sum_{l \neq \{n,m\}} \underbrace{\tilde{\mu}_{\delta_{nl} \to e_n}(m)}_{=0} + \bar{\mu}_{\delta_{nl} \to e_n} + \bar{\mu}_{e_n \to \delta_{nm}} \end{aligned}$$

Therefore $\bar{\mu}_{e_n \to \delta_{nm}} = -\left[s_n(m) + \sum_{l \neq \{n,m\}} \bar{\mu}_{\delta_{nl} \to e_n} \right]$.

Final Equations: Factor to Variable

$$\begin{aligned} \tilde{\mu}_{\delta_{nm} \to e_n}(n) &= \mu_{\delta_{nm} \to e_n}(n) - \bar{\mu}_{\delta_{nm} \to e_n} \\ &= \min[0, \tilde{\mu}_{e_m \to \delta_{nm}}(n)] + \bar{\mu}_{\delta_{nm} \to e_n} - \bar{\mu}_{\delta_{nm} \to e_n} \\ &= \min[0, \tilde{\mu}_{e_m \to \delta_{nm}}(n)] \end{aligned}$$

Final equations for variable to factor

$$\begin{aligned} \tilde{\mu}_{e_n \to \delta_{nm}}(n) &= \mu_{e_n \to \delta_{nm}}(n) - \bar{\mu}_{e_n \to \delta_{nm}} \\ &= \mu_{e_n \to \delta_{nm}}(n) - [s_n(m) + \sum_{l \neq \{n,m\}} \bar{\mu}_{\delta_{nl} \to e_m}] \\ &= s_n(n) + \sum_{l \neq \{n,m\}} \mu_{\delta_{nl} \to e_m} - [s_n(m) + \sum_{l \neq \{n,m\}} \bar{\mu}_{\delta_{nl} \to e_m}] \\ &= s_n(n) - s_n(m) + \sum_{l \neq \{n,m\}} \tilde{\mu}_{\delta_{nl} \to e_m}(n) \end{aligned}$$

15 Some Implementation Details

```
 >>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Fig. 15.1: The Zen, by Tim Peters. Type `import this` at a python command prompt. It summarizes good coding style.

15.1 The Design Goals

Our primary design goals are: *portability, speed, open source* and *ease of use.*

The first decision we had to make was: which programming language? Since our goal was to compare the performance and runtime of different algorithms, we couldn't simply use Matlab because some algorithms need low-level access to the hardware to run efficiently. Our choice was C++ since it is backward compatible to C, has many well-tested libraries available and can be used to write high-performance code. However, C++ has a major drawback: due to its small standard library tasks as for example loading a data file into an array needs already dozens of lines of code. We wanted to avoid coding too much of such high level functions to keep the code clean and simple. Therefore, we decided to use a highlevel language as Python in combination with a "low-level" language as C++. Explicitly, we wanted to run our experiments in the following way.

1. Preprocessing: Use existing and well tested Python packages, for example to load data files with a command like:
   `>>>datapoints = load('mydatafile.txt')`

2. Running the Algorithm: call our C++ functions from Python, for examply by
   `>>>clusterlabels = kmeans(datapoints,K=3)`

3. Postprocessing: Use existing Python packages to process the output and to make plots with an easy command as:
   `>>> plot(x[:,0],x[:,1],'b.')`

As one can see in the Code 15.1, Python perfectly matchs our requirements. Python and C++ are very well supported on all important operation systems. To make our code portable, we have tried to use only the Boost libraries due to their high quality.

Code 15.1. FILE: ap.py
The Python code needed to load the data points positions, compute the similarity matrix, preprocess the similarity matrix, run our implementation of the affinity propagation algorithm and finally output the solution and other information returned by the algorithm. All of that in 18 lines of code! The script is executed on the shell with `$./ap.py`.

```
#!/usr/bin/env python
import numpy as npy
import pylab as pyl
import AffinityPropagation as ap
x = pyl.load('data/ToyProblemData.txt')
N = npy.shape(x)[0]

print 'preparing similarity matrix'
S = ap.outer_dot(x)

print 'putting preferences in the similarity matrix'
median = npy.median(npy.median(S))
P = npy.repeat(median,N)
S[range(N), range(N)] = P
S = -S
print 'running affinity propagation'
dic = ap.ap(S, 100,50,0.5)
print dic
#OUTPUT:
#{'lam': 0.5, 'K': 3, 'it': 58,
#'cl': array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 1, 0, 2, 1, 2, 2, 2, 2, 2, 2, 0, 2,, 1]),
#'dpex': array([ 2,  2,  2,  2,  2,  2,  6,  6,  6,  6, 22,  6,  2, 22,  6, 22, 22, 22, 22,
2, 22, 22,  6]), }
```

15.2 Performance Python

Usually, only small parts of the code turn out to be a speed bottleneck. We show here two approaches that are very useful when speed is an issue: writing a whole extension to Python in a statically typed programming language or using the Weave package. We define a method to be "useful" if it satisfies the following criteria:

- In Python we want work only with standard objects, for example only with Numpy arrays and with Python strings. The underlying C++ must not be visible.
- In C++ we want to work with C++ objects. Examples are the Boost Graph library [Theb] and the Boost multi array library [Thed].
- On the one hand, we need to pass big amounts of data from Python to C++ as reference, on the other hand we need to return the results by value. A typical algorithm may return a list of cluster labels, the estimated centroids and the number of iterations. This should work seamlessly.

15.2.1 Extending Python with C++

Though there are many ways to extend Python, it turned out quite time consuming to find a method that fulfills the above criteria. Among the approaches we considered are

1. Writing a C-extension with the Python API. There you have the full power but the code is ugly! We abandoned the efforts to write native C API extensions since the learning curve is too steep and the code is ugly.

2. Using SWIG to generate the interface automatically. SWIG has proved that it can expose existing C++ libraries to Python. The documentation is quite complete. For our problem however the was no ready-to-use template available. Since SWIG is a very strange mixture of SWIG macros, C++ and Python code, the learning curve is also quite steep and SWIG code is impossible to debug.

3. Using Boost.Python to expose C++ functionality to Python. This approach relies heavily on templated C++ code and therefore needs a standard compliant compiler. Also, the compile time is very high compared to other approaches. The documentation is not the strong point of this library and the examples are not very illustrative either. However, we found a very nice piece of code that solved exactly our problem written by Paul Austin [Thee].

Code 15.2. FILE: ap.hpp

```
#ifndef AP_HPP
#define AP_HPP
#define PY_ARRAY_UNIQUE_SYMBOL PyArrayHandle
#include "num_util.h"

using namespace std;
namespace b = boost;
namespace bp = boost::python;
namespace bpn = boost::python::numeric;
namespace nu = num_util;

bp::dict ap(bpn::array &inSimilaritiesMatrix, uint maxit, uint convit, double lam);

BOOST_PYTHON_MODULE(AffinityPropagation)
{
        import_array();
        bpn::array::set_module_and_type("numpy", "ndarray");
        def("ap",ap);
        def("outer_dot",outer_dot);
}
#endif
```

Code 15.3. FILE: ap.cpp

```
#include <boost/multi_array.hpp>
#include "ap.hpp"

bp::dict ap(bpn::array &inSimilaritiesMatrix, uint maxit, uint convit, double lam){

        /* CHECKING INPUT VALUES */
        nu::check_rank(inSimilaritiesMatrix,2);
        vector<intp> shp(nu::shape(inSimilaritiesMatrix));
        int N = shp[0];
        if(N != shp[1]){
                PyErr_SetString(PyExc_ValueError, "Expected a similarity matrix in (N,N) array form
                bp::throw_error_already_set();
        }

        /* SETUP VARIABLES */
        double* dataPtr = (double*) nu::data(inSimilaritiesMatrix);
        double_matrix_ref       s(dataPtr,b::extents[shp[0]][shp[1]]);  //similarities
        ...
        CODE
        ...
        /* PREPARING OUTPUT TO PYTHON */
        bpn::array ret_dpex     =  nu::makeNum( &dpex[0], N);
        bpn::array ret_cl       =  nu::makeNum( &cl[0], N);

        bp::dict retvals;
```

```
        retvals["K"] = K;
        retvals["lam"] = lam;
        retvals["maxit"] = maxit;
        retvals["convit"] = convit;
        retvals["it"] = it;
        retvals["dpex"] = ret_dpex;
        retvals["cl"] = ret_cl;
        retvals["net_similarity"] = net_similarity;
        retvals["average_preference"] = average_preference;
        retvals["net_self_responsibility"] = net_self_responsibility;
        retvals["net_responsibility"] = net_responsibility;
        retvals["net_availability"] = net_availability;

        return retvals;
}
```

15.2.2 Using the Weave Package

The Weave package comes with the Scipy package [Sci]. It allows to inline C++ code in Python Code. An example can be seen in Code 15.4. At the bottom you can see that loops are incredibly slow in Python! Since it is good coding style not to use loops, this is usually not a big issue. Sometimes, however, it is either impossible or inconvenient to use high-level functions to perform some low-level array operations. Then it's very nice to write a little C-extension in 5 lines of code.

Code 15.4. FILE: matrix_speed/ms.py

```
#/usr/bin/env python
import numpy as npy
import time
import scipy.weave as weave

N = 2000

# two for loops
print 'starting testrun: two for loops'
t_start = time.time()
A = npy.zeros((N,N),dtype=float)
for n in range(N):
        for m in range(N):
                A[n,m] = N*n + m
t_end = time.time()
print 'time in seconds: %f'%(t_end-t_start)

# numpy style
print 'starting testrun: numpy style'
A = npy.zeros((N,N),dtype=float)
t_start = time.time()
A = npy.asmatrix([[ N*n + m for m in range(N)] for n in range(N)])
t_end = time.time()
print 'time in seconds: %f'%(t_end-t_start)

#weave style
A = npy.zeros((N,N),dtype=float)
code = """
for (int n=0; n!=N ; ++n) {
        for (int m = 0; m!=N; ++m){
        A(n,m) = N*n + m;
        }
}
return_val = 0;"""
print 'starting testrun: weave - inlined c++ code'
t_start = time.time()
weave.inline(code, ['A','N'],type_converters=weave.converters.blitz,compiler = 'gcc',verbos
```

```
t_end = time.time()
print 'time in seconds: %f'%(t_end-t_start)
#OUTPUT:
#starting testrun: two for loops
#time in seconds: 4.616837
#starting testrun: numpy style
#time in seconds: 3.917110
#starting testrun: weave - inlined c++ code
#time in seconds: 0.056107
```

15.3 Some Remarks about Debugging Algorithms

This little subsection is a little collection of things we learned about writing algorithms. Though this section is very short, we spent at least 50% of the total time on identifying bugs and acquiring the know-how to speed up the debugging process.

- Check first if the theory of an algorithm is correct. One can test the theory by visual inspection,comparison to a reference, and most importantly, by making the complete derivation of the algorithm; and not solely relying on papers and text-books. A refereed journal paper does not guarantee that the algorithm works as wanted. In our case, we could not get deterministic annealing to work properly with the algorithm explained by Rose [Ros98]. Also, there is much confusion in the naming of methods. Ward's method, for instance, does not optimize the K-means objective function, though it is stated in the literature that it does.

- Check the most likely bug first. This seems obvious, but when we experienced random errors in our algorithms with errors as "glibc detected invalid Python free" as we wrote Python extensions, we were afraid we spent weeks on acquiring the know-how to extend Python only to find out that we can't use it! However, in the end, a Python free error was simply a buffer overflow. The reason why much time was spent to verify that the external libraries we decided upon work correctly is the following: The available tools for debugging C++ code under Linux are quite poor: To our knowledge, there is no free debugger available that allows the introspection of STL containers (vectors, lists,...). We failed to get the debugger to work on C++ code that is called from a Python process.

- Check the bug that is easiest to fix first.

- Prototype in Python! What may be done in Python in two hours may need days in C++, or more. In our experience, the best approach to implement an algorithm in C++ is the following: Prototype in Python and pipe the output to a file. Rewrite the algorithm in C++. Of course, the first time, the C++ code is called, the algorithm won't work. Using a program as the KDE program `kompare` allows to compare the outputs of the Python implementation and C++ implementation line by line. This way one can find very quickly where the algorithm failed.

16 Clustering in High Dimensions

Analyzing systems of high dimensionality is hard since the high dimensionality does not allow visual introspection and systems may behave completely differently from one dimension to another. In this section, we show two simple experiments that indicate how the pairwise distances between two clusters behave when the dimension is increased. In both experiments there are two clusters. The data is generated as follows:

Code 16.1. Experiment of Figure 16.2:

```
#draw data points from a normal distribution centered around [-10,-10,...,-10] and [+10,+1
y = npy.zeros((2,D),dtype=float)
y[0,:] = -10
y[1,:] = 10
#draw data points from a normal distribution centered around  [-10,-10,...,-10]
x0 = rand.normal(0,1,size = (N,D))
x0 += y[0,:]
#draw data points from a normal distribution centered around [+10,+10,...,+10]
x1 = rand.normal(0,1,size = (N,D))
x1 += y[1,:]
#combine results
x = npy.append(x0,x1,axis=0)
```

Code 16.2. Experiment of Figure 16.1:

```
#draw data points from a normal distribution centered around [-10,0,0,...,0] and [+10,0,0,
y = npy.zeros((2,D),dtype=float)
y[0,0] = -10
y[1,0] = 10
#draw data points from a normal distribution centered around [-10,0,0,...,0]
x0 = rand.normal(0,1,size = (N,D))
x0 += y[0,:]
#draw data points from a normal distribution centered around [+10,0,0,...,0]
x1 = rand.normal(0,1,size = (N,D))
x1 += y[1,:]
#combine results
x = npy.append(x0,x1,axis=0)
```

The results can be seen in Figure 16.1 and 16.2. From the clustering perspective, it would be nice if the data in high dimensional spaces behaved as in 16.2. However, the centroid is usually located on a submanifold of low dimension which is embedded in a high dimensional space. For example, consider the case where two cluster centroids are located on a $2D$ hyperplane embedded in a 1000 dimensional space. This corresponds to Figure 16.1.

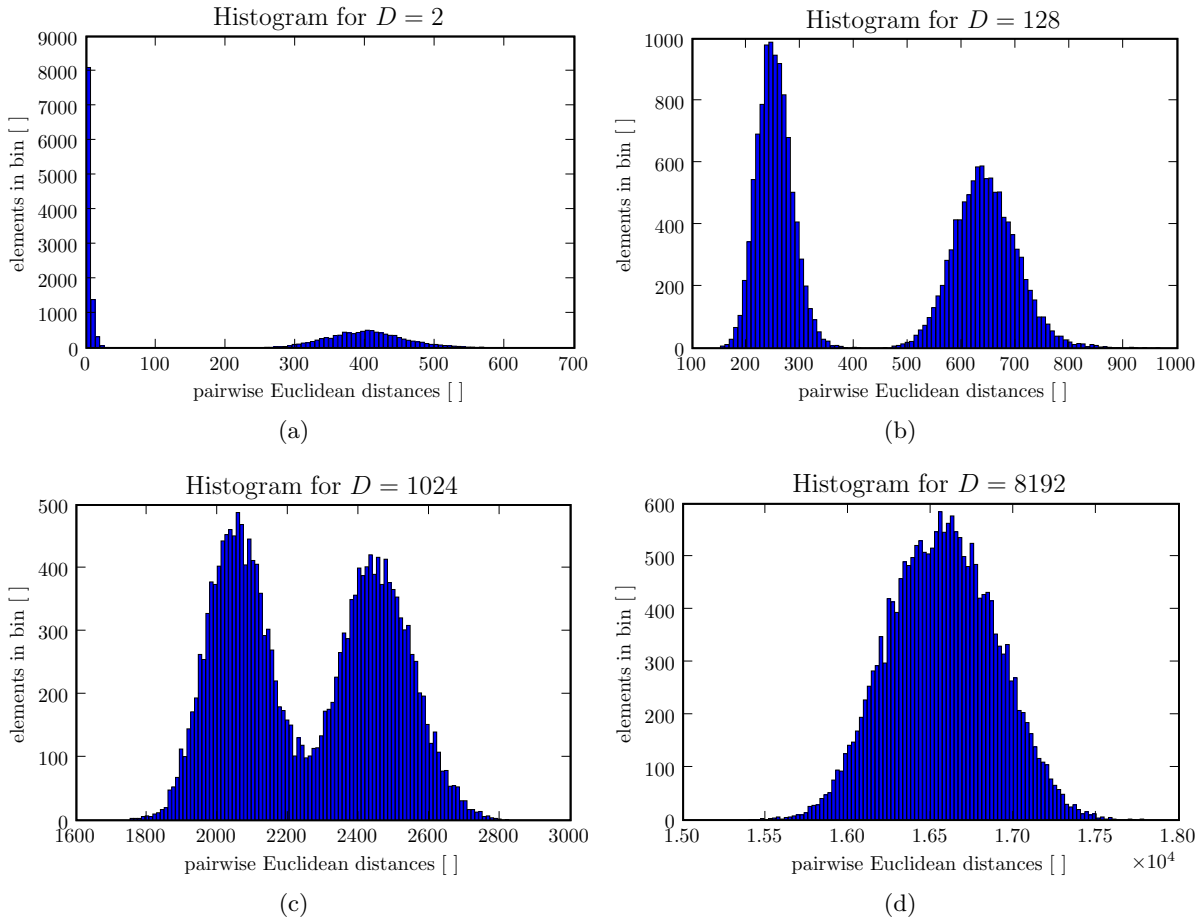


Fig. 16.1: The two cluster centroids are located at $[-10, 0, \ldots, 0]$ and $[10, 0, \ldots, 0]$. When the dimension D is increased, one cannot distinguish the clusters based on their pairwise distances, though the clusters are separated.

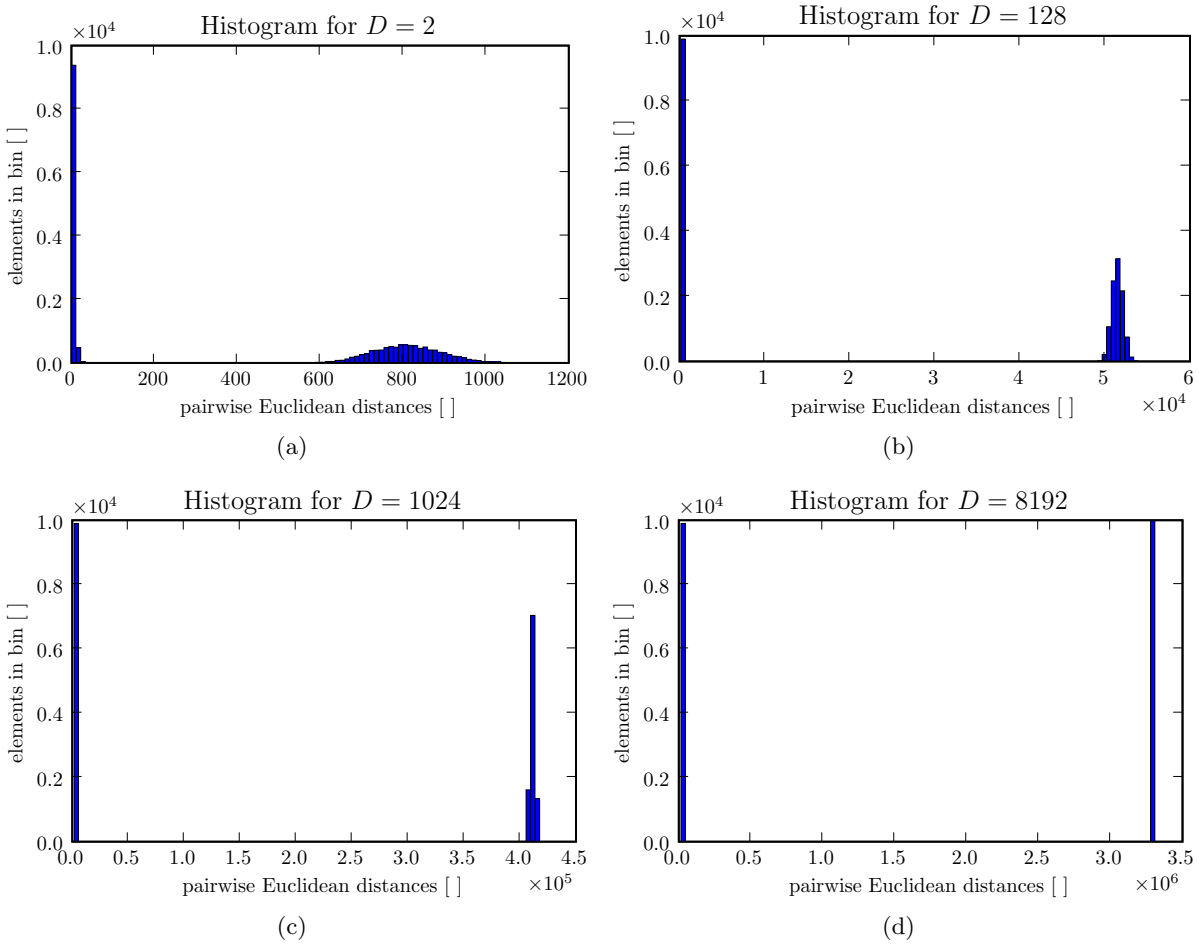
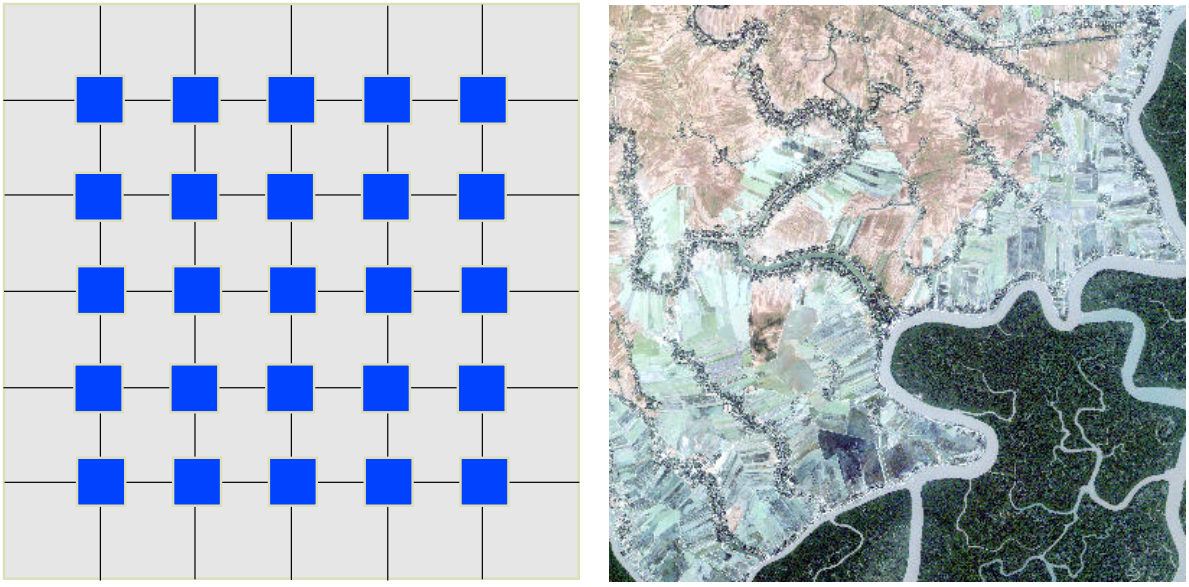


Fig. 16.2: The two cluster centroids are located at $[-10, -10, -10, ...]$ and $[10, 10, 10, ...]$. When the dimension D is increased, the clusters remain easily seperable.

17 Color Image Segmentation by Histogram Clustering

Since toy datasets are generated by an experimentator, they are likely to be biased. This can lead to wrong conclusions about the performance of an algorithm. It is therefore important to test the algorithms on real datasets. We choose to use datasets as they arise in the problem of image segmentation. After fruitless attempts with texture image segmentation we resorted to color image segmentation by histogram clustering. The steps in *histogram clustering* are:

1. Subsample the image, e.g., use only every 20'th pixel in x and y direction.
2. Read the color values of neighboring pixels. For example use a 11×11 box to obtain 121 intensity values for each color. (See Figure 17.1.)
3. Make a histogram. For example use 8 bins for each color. For each sampled pixel, one obtains a histogram with 24 bins. The difference to a 24 dimensional vector is that the sum of elements of all bins is the same for all histograms. The histograms should be normalized to a certain constant, since in soft assignement ACM, the normalization affects the result.
4. Define a distance measure between the data points. For example the Kullback-Leibler divergence.
5. Run a clustering algorithm.

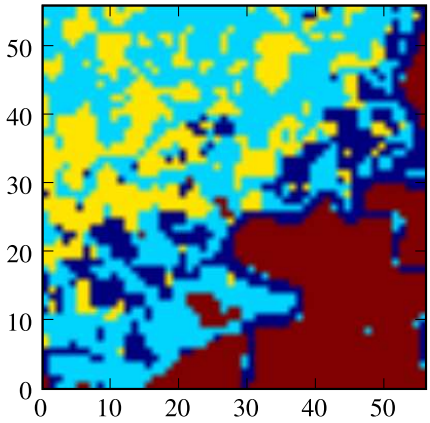


(a) (b) Original image.

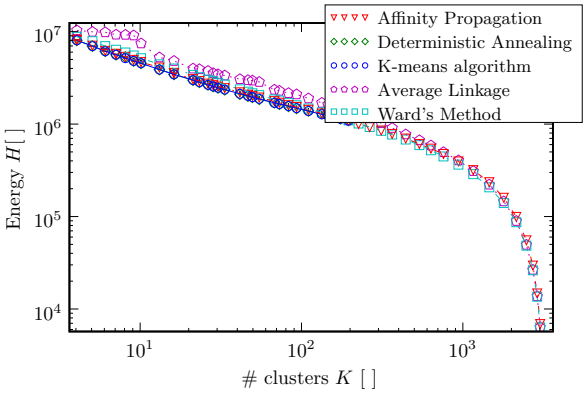
Fig. 17.1: (a) shows how the data points are generated. At each intersection of two lines a datapoint is generated as follows. For each color channel read the intensities of the pixels in the box around the intersection and make a histogram with 8 bins. Therefore each datapoint results in a histogram with 24 bins. (b) is a Quickbird satellite image that we use for the comparison.

17.1 Performance Comparison of hard/soft ACM vs AP

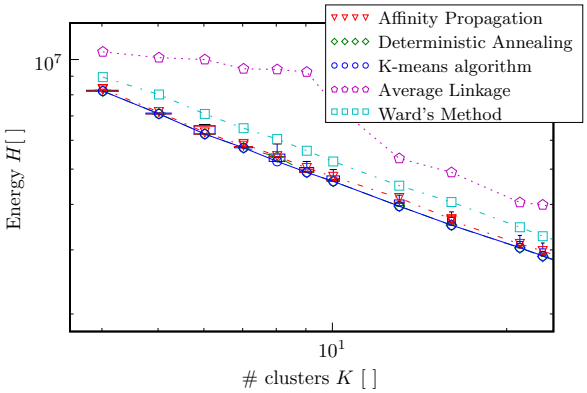
We were curious how AP (c.f. Algorithm 10 performs on similarity matrices that are not symmetric. As example, we picked the problem of histogram clustering with the negative Kullback-Leibler divergence D^{KL} as similarity measure. The objective function is defined in Eqn. (3.11). As one can see in Figure 17.4, soft assignment ACM (c.f. Algorithm 7) does not perform too well compared to AP and hard assignemnt ACM (c.f. Algorithm 6). However, since ACM is a non-zero temperature variant of hard clustering, the objective function that soft assignment ACM tries to optimize is the free energy and not the energy H ($U = H$ for $T = 0$).



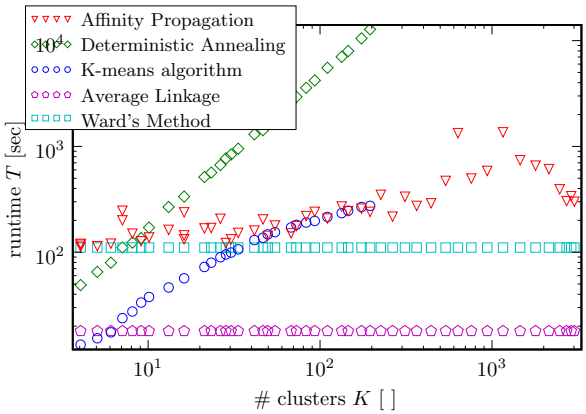
(a) Image segmentation found by affinity propagation. The similarity measure is the squared Euclidian distance.



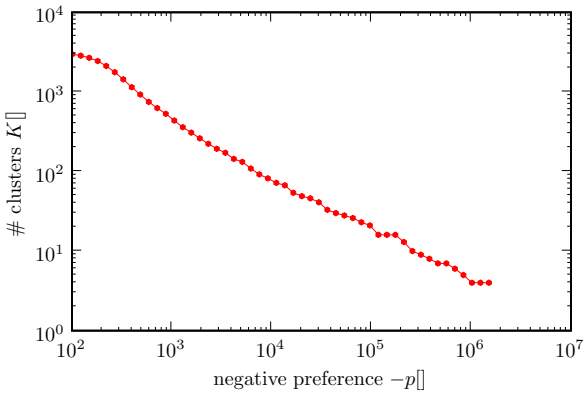
(b) The performance of the clustering algorithms.



(c) The performance of the clustering algorithms.



(d) The runtime analysis.



(e) Affinity propagation's internal cluster validation heuristic.

Fig. 17.2: We use the different algorithms to segment a Quickbird satellite image. Due to memory problems of the affinity propagation algorithm the image is only very coarsely sampled. Affinity propagation gives good results but needs much more time than the K-means algorithm or DA. Average linkage gives much worse results than the other algorithms.

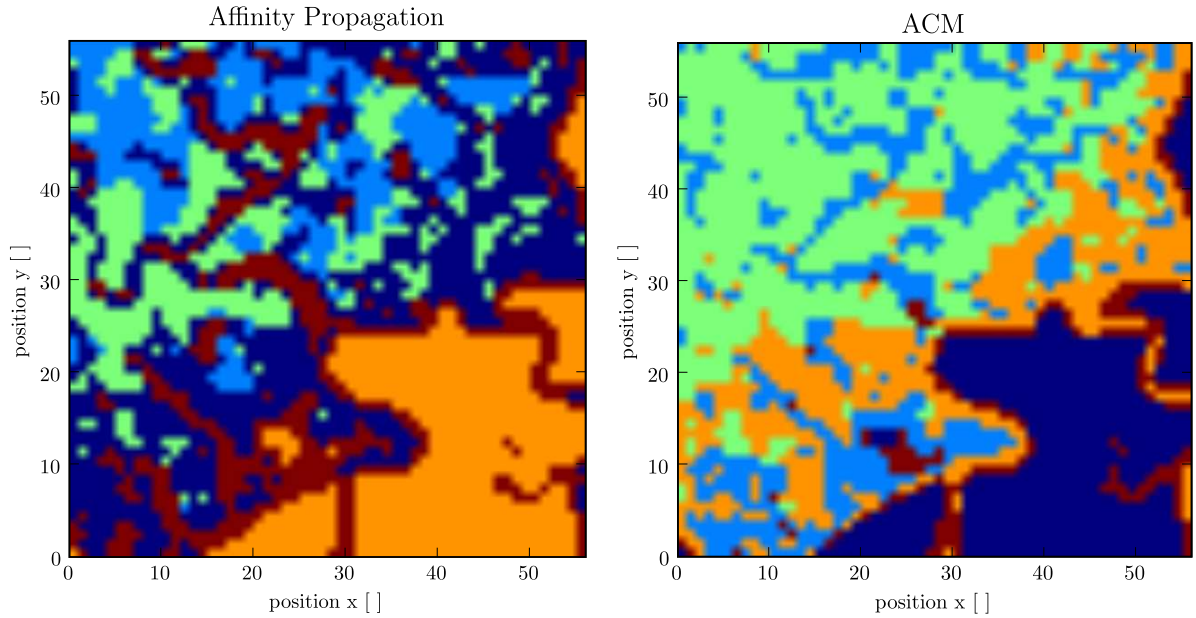


Fig. 17.3: (a) The image segmentation solution found by affinity propagation with the negative Kullback-Leibler divergence as similarity measure. In (b) we depicted a solution found by ACM. We picked the best of 10 runs by visual inspection.

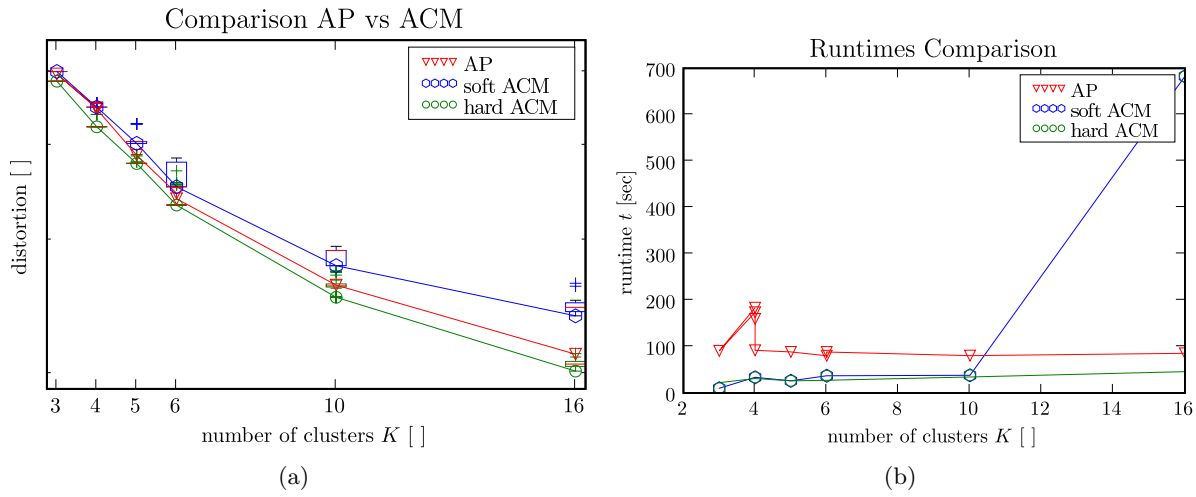


Fig. 17.4: A performance comparison between ACM and AP. The measurements with hexagons (blue) correspond to ACM while the triangles (red) correspond to AP. One can see that AP finds better results than ACM if the objective function is assumed to be Eqn. (3.11). The y-label "distortion" is another name for the objective function H^{ACM}. Similarly to the K-means problem, when the numbers of clusters is low, a simple iterative algorithm can outperform AP. We restarted ACM 200 times to get a feeling how ACM performs in average. The big difference in the performance between ACM and AP is probably due to the fact that they optimize different objective functions. We do not have a good explanation for the kink that can be seen in the runtimes of the soft assignment ACM algorithm.

18 Finding Exons by Clustering Microarray Data

An application of clustering can be found in biological and medical research. Here we show how affinity propagation has been used to find genes resp. exons in the mouse genome ([FD07b, FMM+05]). To understand the objective of the clustering, we give a short summary of molecular biology ([CR05, Cla06]).

18.1 The Molecular Biological Background of Genes

All the information that is needed to grow an organism is saved in the *deoxyribonucleic acid* (DNA). The DNA is a sequence of four nucleic acids: *adenine* A, *guanine* G, *cytosine* C, *thymine* T. In the cell the DNA sequence is found as one strand of the double helix. The two strands are complementary to each other (see Figure 18.2 for clarification). Though the DNA sequence contains all the information about an organism, it is not the build plan for it. It is rather the build plan for its components (proteins) and its builders (enzymes, transport proteins, regulatory proteins, tRNA). Not all of the DNA sequence is of direct relevance for the production of the components and builders: only subsequences. A *gene* is a subsequence of the DNA that produces functional RNA. The sequence is located between an *initiation codon* which defines the start of the gene and an *termination codon*. If the RNA is used to build a protein, the gene is called a *protein-coding gene* and otherwise a *non-protein-coding gene*.

The production of a polypeptide in eukaryots needs three steps: The transcription, the splicing and the translation.

The transcription makes a complementary *messenger RNA* (mRNA) copy of a gene located in a strand of the double helix. RNA is the *ribonucleic acid* and consists of a usually one stranded sequence of the four nucleic acids A, U, C, G, i.e. in comparison to DNA T is replaced by *uracil* U. Therefore, A is complementary to U (A $\leftrightarrow$ U) and C $\leftrightarrow$ G. It is called messenger RNA, because in eukaryotic cells this RNA transports the information of the DNA from the nucleus to a ribosome site where the polypeptide is built.

The Splicing In eukaryots (animals,plants,fungi,...) not all of the gene's DNA sequence is needed to produce a polypeptide. There are so-called *introns* that are removed from the mRNA sequence in a process called *splicing*. The fragments between the introns that are not spliced away are called *exons*. They are possibly coding sequences. After the splicing, the exons build a new sequence of mRNA. The exons usually make up only a small part of the gene (usually about 10%). To distinguish those two variants, one calls unspliced mRNA the *pre-mRNA*. See Figure 18.1 for a graphical representation of a RNA sequence containing introns and exons.

The translation is the synthesis of polypeptides according to the 'build plan' saved in the mRNA. Basically this is the translation of the information saved as RNA (i.e. the sequence of the four letters A,U,C,G) to a sequence of the 20 amino acids of which the polypeptides are built of. This is done by the *ribosome* which is a complicated structure of many proteins and nucleic acids. The *triplet code theory* states that three successive RNA letters uniquely define one amino acid. For clarification see Figure 18.4 and 18.5.

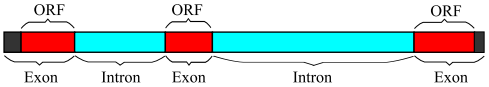


Fig. 18.1: One can see a graphical representation of an RNA sequence. In the splicing, the introns are removed. One can also see that parts of the exons are possibly non-coding and do not belong to the open reading frame (ORF).

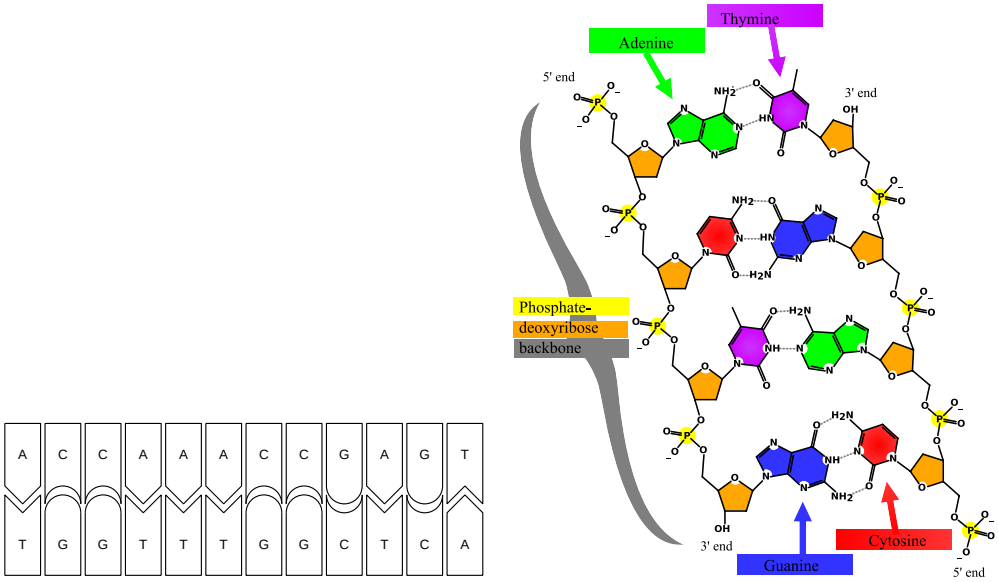


Fig. 18.2: a)The nucleic acids are complementary to each other. A ↔ T and C ↔ G. Other combinations are not possible because the chemical structure does only allow bonds between the complementary nucleic acids. The bonding of the molecules is depicted in b). The *3' end* and *5' end* that can be seen in the graphical representation in b) are important because together with the promoter which defines where the starting point is, they define which strand is used to produce the mRNA (the other strand is a complementary copy and not an exact copy and would therefore lead to another mRNA sequence).

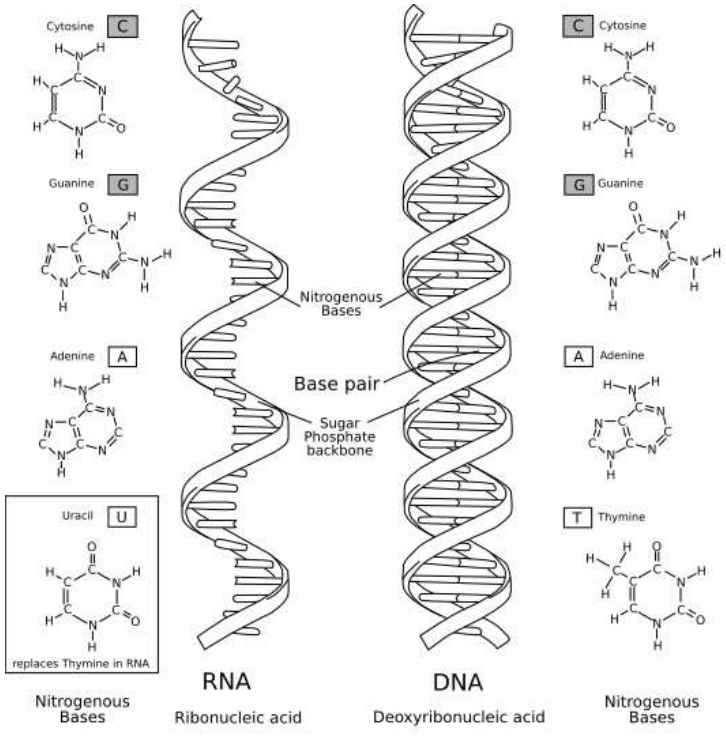


Fig. 18.3: A comparison between DNA and RNA.

18.2 Clustering Microarray Data

As pointed out above, most of the DNA is not of direct relevance: Only genes are interesting since they produce functional RNA. The job is to find genes! One way to do that is using

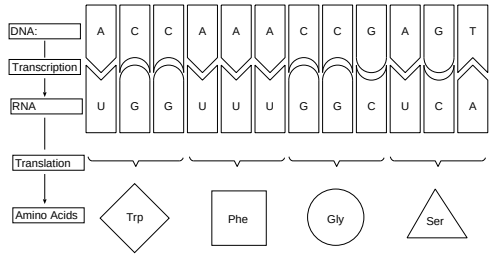


Fig. 18.4: In the transcription a complementary mRNA copy of the DNA is produced. The mRNA is then translated by a ribosome to a sequence of amino acids (which is a polypeptide). Here one can see that for example the DNA sequence ACC is transcribed to UGG which is translated to Trp.

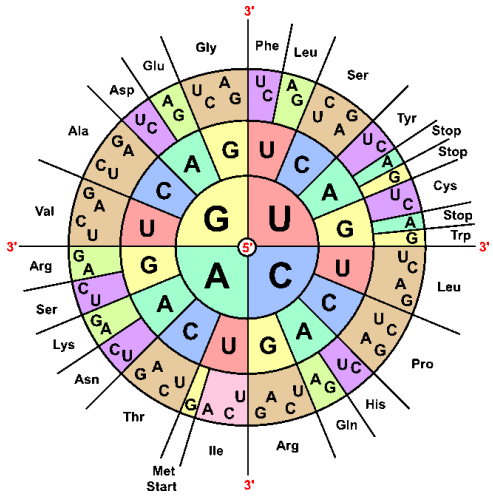


Fig. 18.5: The *codon sun* is a graphical representation of the translation process. For example, going from the center to the edge, UUU and UUC are translated to the amino acid Phenylalanine.

so-called *micro arrays* which are also known as *DNA chips* since they look a bit like computer chips (see Figure 18.6) and are also produced similarly by lithography. The fundamental idea is the following: The DNA is the same in all cells of an organism. For example a liver cell has the same DNA as a heart cell. The difference between the cells is only that a complex interaction of proteins and enzymes that float in the carioplasm make the genes in the DNA to be expressed to different extents. It is assumed, that genes are active in all cells, just more active in some and less in others. If one knew that in most cells there is a certain mRNA sequence, one could conclude that it was produced by a gene (and not by some other process) and that the mRNA sequence is a complementary copy of an exon. Thus, the first step to find new genes is to find the exons.

The steps needed to find exons with microarrays are:

1. Find sequences that are putative exons. This can be done by the use of prior information and heuristic algorithms (for example GenScan, HMMGene, GrailEXP or BlastX) that make predictions solely based on the DNA sequence.

2. Create many copies of a certain DNA fragment that is attached to the chip.

3. Take mRNA samples from many different cells and mark them with a fluorescent dye.

4. Put mRNA sample on a spot. If the sequences of the DNA fragment and the mRNA are complementary, they *hybridize* and therefore the mRNA is also attached to the chip.

5. Remove non-hybridized mRNA.

6. Turn on UV light to see if there was hybridization.

A microarray can have up to 100,000 spots with certain DNA fragments mounted on a square centimeter of the chip. The minimal length of the fragment is around $l = 16$ because in a random sequence with 4 letters, the expected frequency of the fragment is $1/4.2950e + 09$. This approximately the size of a genome. Usually the fragments have 25 to 60 bases to prevent random and cross hybridization.

In the mouse genome experiment the DNA length was 60 bases and tissue samples from 12 different cells of a mouse have been taken. The hybridization of each tissue sample is then checked on 78,000 fragments that belong to putative exons. One then gets a result as shown in Figure 18.6. The intensity of the light corresponds (usually not linearly and with a lot of noise) to amount of a certain mRNA sequence in the cytoplasm.

With this information it is possible to find genes. One defines a similarity measure $s(n, m)$ between two DNA sequences n and m. If the sequence of n and m is far apart in the DNA sequence, it is very unlikely that they belong to the same gene. If they are close in the DNA sequence but they have different expression levels in the tissue samples, they are also unlikely to belong to the same gene. The should be grouped to the same gene iff they are nearby and show transcription across multiple tissues. Since affinity propagation assigns each data point n an exemplar, even if it is actually an intron, one has to add an additional exemplar. The idea is that all introns connect to this exemplar. The similarity measure is probably defined as [FD05]

$$s_{nm} = \log\left\{\lambda e^{-\lambda|n-m|}\left(qp_0(x_n) + (1-q)\int p(y,z,\sigma)\frac{e^{-\frac{1}{2\sigma^2}\sum_{d=1}^{D}(x_{md}-(yx_{md}+z))^2}}{\sqrt{2\pi\sigma^2}^D}\mathrm{d}y\mathrm{d}z\mathrm{d}\sigma\right)\right\}, \tag{18.1}$$

where x_{nd} is the expression level of the d'th tissue in the n'th probe. The probes are sorted in genomic order. Therefore, the exponential $e^{-\lambda|n-m|}$ results in a cutoff for large distances, where s_{nm} tends to $-\infty$. λ is a parameter that can be varied. $p_0(x_n)$ is a background distribution that accounts for false putative exons. q is the probability of a false putative exon within a gene.

One ends up with a $78,000 \times 78,000$ similarity matrix with many similarities that are $-\infty$ because they are too far apart in the DNA sequence. The matrix is therefore sparse. In the experiment, the similarity is measured by

organism	genome size [bases]	estimated number of genes [ ]	genes per base [ ]
H. influenzae (bacterium)	1.8 Mb	1700	950
S. cerevsiae (yeast)	12 Mb	6000	500
A. thaliana (plant)	97 Mb	19000	200
S. melanogatster (fly)	180 Mb	13000	100
H. sapiens (human)	3200 Mb	20488 – 20588?	10 – 20

Tab. 18.1: The number of bases in the genome of different species. One Mb is one million base pairs. One base needs 2 bits and therefore the complete genome of humans can be saved in about 800 MegaByte. While the DNA sequence is known, the number of protein coding genes is still an open question. In 2000 the estimates ranged from 28,000 to 150,000 protein coding genes. In May 2007 the estimate is estimated around 20488 [Pen07]. There is a correlation between the complexity of an organism and the size of the genome. However, there are examples where seemingly simple organisms have a much larger genome than humans.

18.3 Using AP to Identify Exons

We took the similarity matrix provided on Frey's homepage [FD07b] and ran affinity propagation with different input preferences. The output of AP gives us the information:

1. Is the putative exon an exon?

2. To which gene do the exons belong.

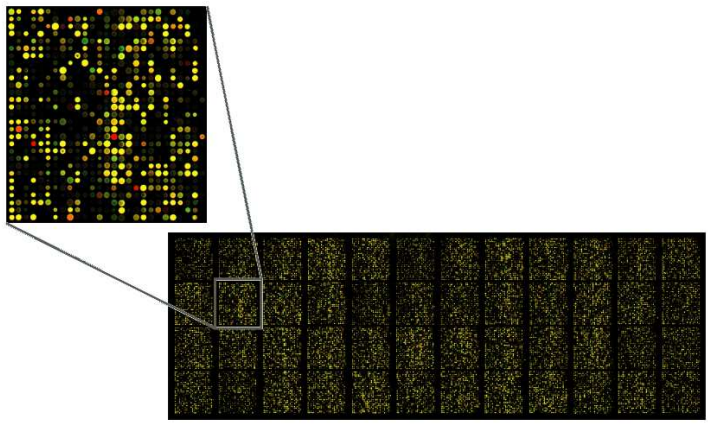


Fig. 18.6: A DNA chip. In each dot many copies of a certain DNA sequence with 15 to 60 bases are fixed to the glass plate. When mRNA complementary to the DNA sequence is added, the DNA sequence and the mRNA hybridize: The mRNA is then also fixed to the glass plate. The mRNA is labeled with a fluorescent dye. When mRNA that has not hybridized is removed, only those dots are fluorescent where the sequences matched. The DNA chip is of smaller size than presented here. The top left part is smaller than $1cm \times 1cm$ in size. Currently, up to 100 000 spots fit on a square centimeter. The colors red and green come from different different fluorescent dyes that have been used to tag the mRNA samples. Yellow is the superposition of red and green. That way one can use one spot to test for mRNA taken from two samples. This trick allows to test the gene expression from different samples.

To check the quality of the result, Frey computes the true-positive rate and the false-positive rate of the clustering solution by a comparison to a ground truth dataset. The ground truth is also available on Frey's homepage. The dataset consists of a sequence of known exons and a sequence of known introns. Explicitly, if it is known that data point n is really an exons, the n'th position of a binary vector of length N one and zero otherwise. Analogously for introns. We give here the Python code that computes the true positive rate and the false positive rate:

```
#ap_exon: binary vector of length N,  1 if AP classified as exon, 0 else
#refseq_exon: reference vector of known exons
#refseq_intron: reference vector of known introns
#computing true-positive and false positive rate, Python multiplication of arrays is elemen
true_positive_rate = sum( ap_exon * refseq_exon) / sum(refseq_exon)
false_positive_rate = sum(ap_exon*refseq_intron) / sum(refseq_intron)
```

Note, that the true-positive rate and the false positive rate do not have to add up to one since they are computed under other conditions, i.e., the true-positive rate is

$$p(\text{classification as exon}|\text{is exon}, \text{known exons}) \tag{18.2}$$

and

$$p(\text{classification as exon}|\text{is intron}, \text{known introns}) \tag{18.3}$$

the false-positive rate is. In the limit, where all putative exons are classified as exons by AP, the true-positive rate would be one. On the other hand, at the same time, the false-positive rate would also tend to one. The goal is a high true-positive rate at low false-positive rate. This is a trade-off problem as one can see in Figure 18.7.

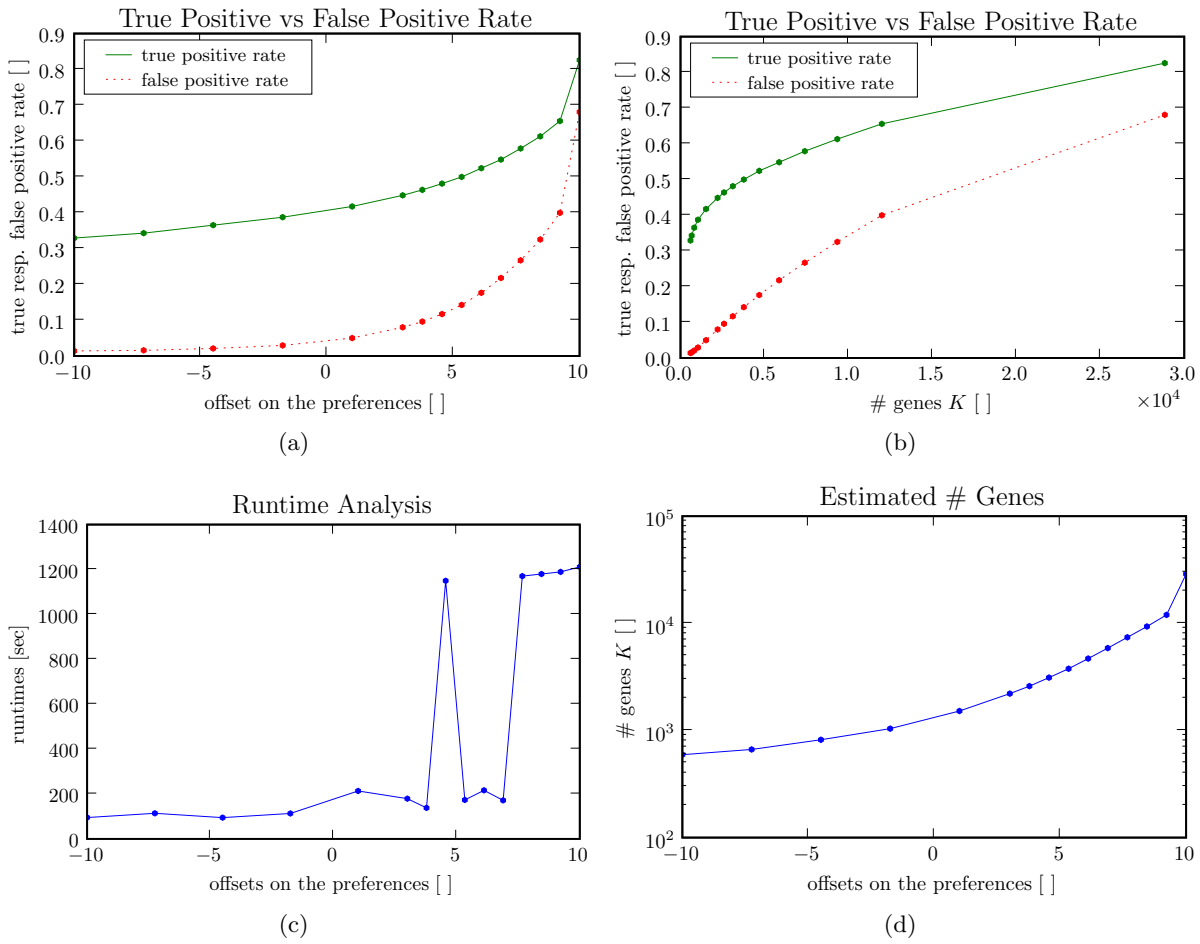


Fig. 18.7: The result of the affinity propagation algorithm restarted with several offsets on the preferences. In (a) and (b) one can see, that high true-positive rate comes with a high false-positive rate. (c) shows the runtime of AP. The high values correspond to cases, where the algorithm did not converge; the results are plausible anyway. (d) shows, that there is no plateau that would indicate the *right* number of clusters.

19 Experiments on Toy Datasets

Experiments with few clusters show that the K-means algorithm works sufficiently well [data not shown]. The reason is twofold: On the one hand, each iteration of the K-means algorithm needs $\mathcal{O}(K)$ arithmetic operation. When K is small, the algorithm runs quite fast. On the other hand, few clusters mean, that it is likely that at least one restart of the K-means starts with one centroid in each cluster.

In this section, we investigate the properties of the algorithm for many clusters and high dimensionality. The experiments show, that the $\mathcal{O}(K)$ property of the K-means algorithm and DA can lead to very high running times. Furthermore, with many clusters, the chance that K-means starts with one centroid at each cluster can get very small. Then, many restarts are necessary to find the optimal solution. Possibly, this deficiency could be overcome by bagging.

19.1 How the Datasets are Generated

We show here Python code that has been used to generate the toy datasets. The function `create_gaussian_dataset()` is not shown.

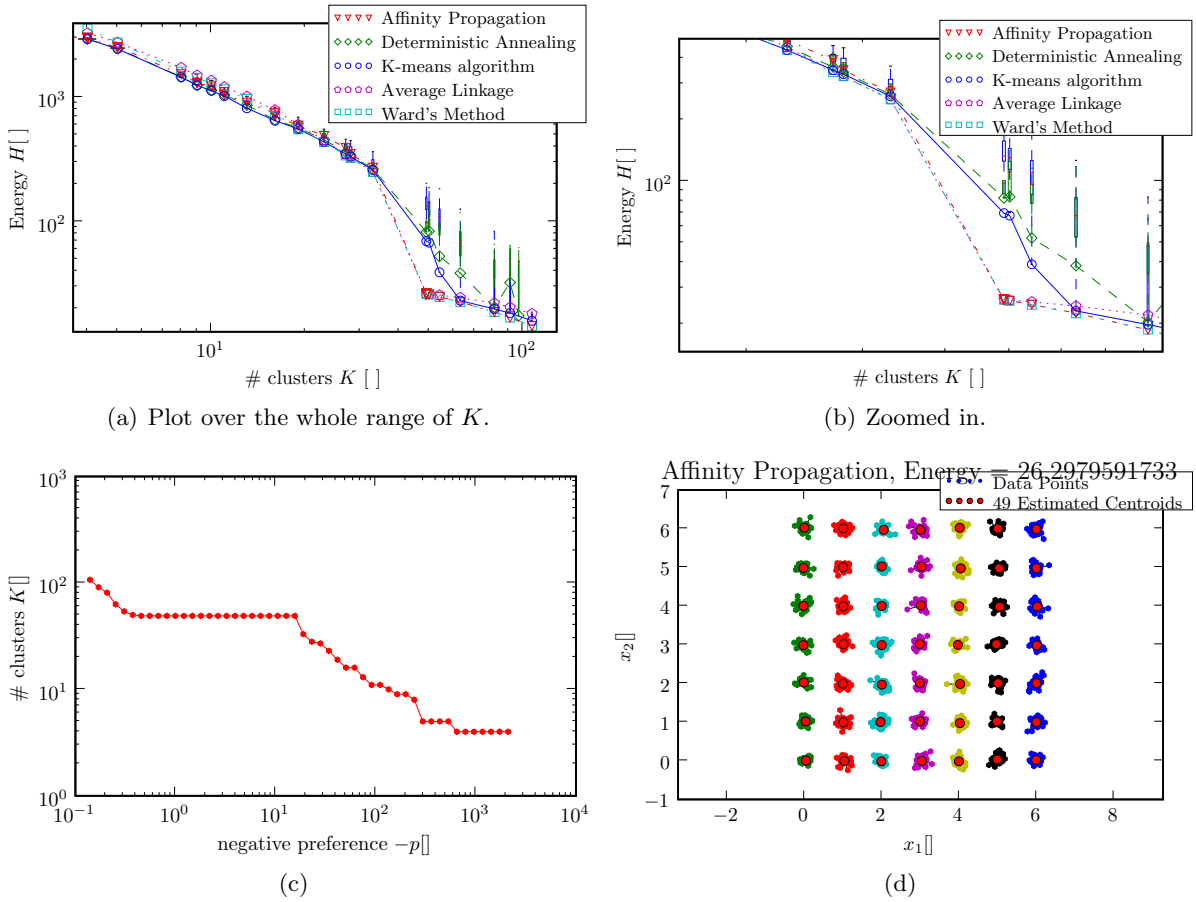
```
#-------------------------------------------------------------------------
# WELL DEFINED CLUSTERS ON A 2D GRID DIFFERENT CLUSTER SIZES
#-------------------------------------------------------------------------
dataset_name = 'DiffClusterSizeWD'
```

```
K = 7; K2 = K**2; D = 2
y = npy.zeros((K2,D),dtype=float)
for k in range(K2):
        y[k,:] = [k%K,k/K]
variances = 0.1 * npy.ones(K2,dtype=float)
Ns = rand.randint(0,60,(K2,))
for N_noise in [0,500]:
        create_gaussian_dataset(y,variances,Ns,dataset_name, N_noise)

#-----------------------------------------------------------------------
# WELL DEFINED CLUSTERS ON A 2D GRID DIFFERENT SAME CLUSTER SIZES
#-----------------------------------------------------------------------
dataset_name = 'Grid2DSameClusterSizeWD'
K = 7; K2 = K**2; D = 2
y = npy.zeros((K2,D),dtype=float)
for k in range(K2):
        y[k,:] = [k%K,k/K]
variances = 0.1 * npy.ones(K2,dtype=float)
Ns = 30*npy.ones(K2,dtype=int)
#create dataset with different levels of noise
for N_noise in [0,500]:
        create_gaussian_dataset(y, variances, Ns, dataset_name, N_noise)

#-----------------------------------------------------------------------
# RANDOM WELL DEFINED CLUSTERS IN SMALL DIMENSIONS WITH EQUAL VARIANCES
#-----------------------------------------------------------------------
dataset_name = 'RandDiffClusterSizeWD_noisy'
D = 2
for K in [5,10,20,50,100,150]:
        Ns = rand.randint(0,20,K)
        y = 10.* rand.rand(K,D)
        variances = 0.5/(K**(1./D)) * npy.ones(K,dtype=float)
        for N_noise in [0,100,500]:
                create_gaussian_dataset(y, variances, Ns, dataset_name, N_noise)

#-----------------------------------------------------------------------
# RANDOM WELL DEFINED CLUSTERS IN SMALL DIMENSIONS WITH DIFFERENCENT VARIANCES
#-----------------------------------------------------------------------
dataset_name = 'RandDiffClusterSizeDiffVarWD_noisy'
D = 2; K = 50
Ns = rand.randint(0,60,K)
y = 10.* rand.rand(K,D)
variances = 2./(K**(1./D)) * rand.rand(K)
for N_noise in [0,500]:
        create_gaussian_dataset(y, variances, Ns, dataset_name, N_noise)

#-----------------------------------------------------------------------
# WELL DEFINED CLUSTERS GROWING DIMENSIONS
#-----------------------------------------------------------------------
dataset_name = 'GrowingD_WD'
Ds = [2,4,8,16,32,64,128,256,512]; K = 50
for D in Ds:
        Ns = rand.randint(0,20,K)
        y = 10*rand.rand(K,D)
        variances = 0.2 * rand.rand(K)
        create_gaussian_dataset(y, variances, Ns, dataset_name, N_noise)
```

19.2 Example on a Grid in 2D - Same Cluster Sizes - Same Variances

This experiment is designed to investigate if the K-means algorithm can find the global optimum when the clusters are nearby but well-defined. As one can see in 19.1, the K-means algorithm does not perform very well in this setting. AP has not problem to find a good solution. We wondered how additional noise affects the algorithms. As one can see in Figure 19.2 the result is qualitatively the same. The dataset is generated as explained in 19.1 `# WELL DEFINED CLUSTERS ON A 2D GRID DIFFERENT SAME CLUSTER SIZES`.

19.2.1 Without Noise



(a) Plot over the whole range of K.

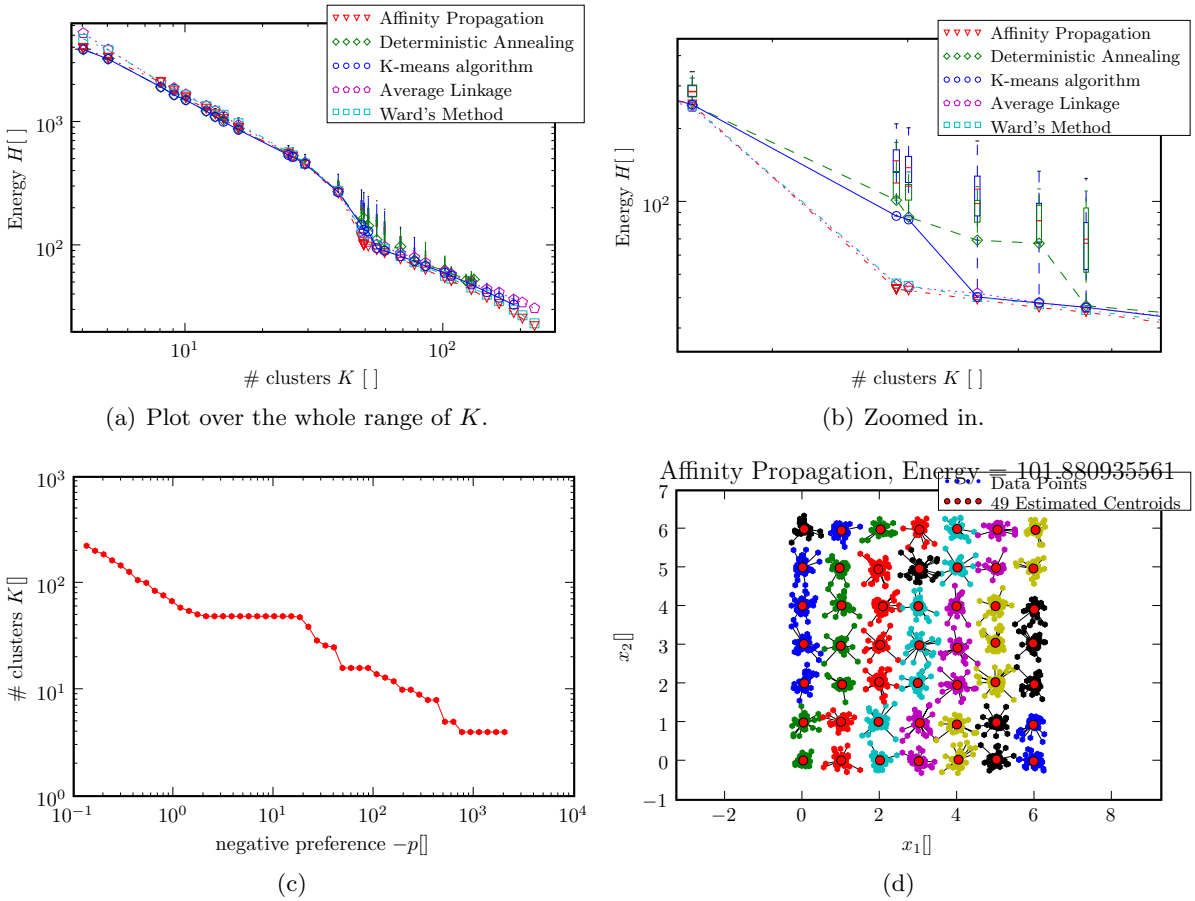
(b) Zoomed in.

(c)

(d)

Fig. 19.1: Example On Grid in 2D - Same Clustersizes - Same Variances, no noise. $D = 2$, $K = 49$, $N = 1470$, $N_{\text{noise}} = 0$. K-means is restarted 400 times, DA 20 times. In (a) and (b) one can see that Ward's method and AP work well on this dataset. Especially near the true number of clusters $K = 49$, they perform much better than the K-means algorithm and DA. In (c) one can see, that AP's internal cluster validation correctly identifies 49 clusters since there is a big plateau. (d) shows the clustering solution found by AP with $K = 49$.

19.2.2 With 500 Noisy Data Points



(a) Plot over the whole range of K.

(b) Zoomed in.

(c)

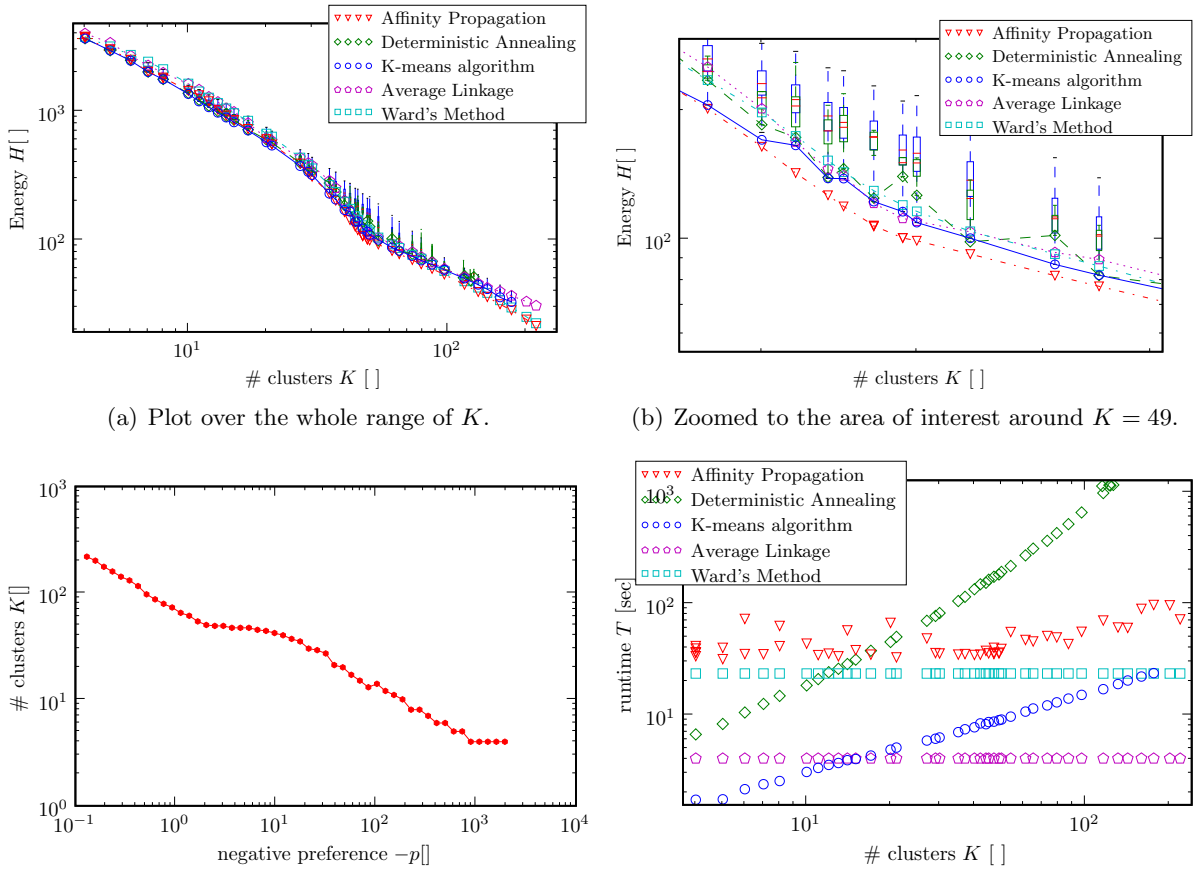
(d)

Fig. 19.2: Example On Grid in 2D - Same Clustersizes - Same Variances. $D = 2$, $K = 49$, $N = 1470$, $N_{\text{noise}} = 500$. K-means is restarted 400 times, DA 20 times. The interpretation is the same as above.

19.3 Example on a Grid in 2D - Different Cluster Sizes - Same Variances

The dataset is generated as explained in 19.1 `WELL DEFINED CLUSTERS ON A 2D GRID DIFFERENT CLUSTER SIZES`.

19.3.1 With 500 Noisy Data Points



(a) Plot over the whole range of K.

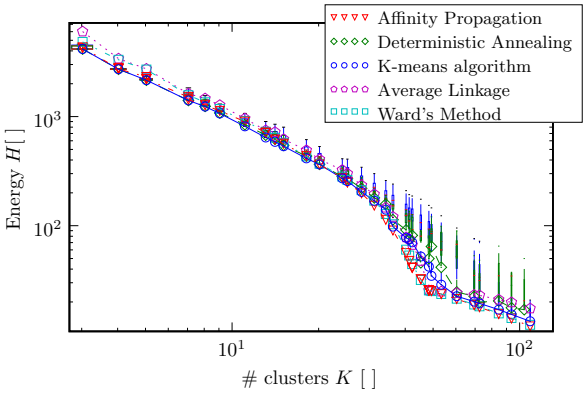
(b) Zoomed to the area of interest around $K = 49$.

(c) There is a plateau at $K \approx 50$.

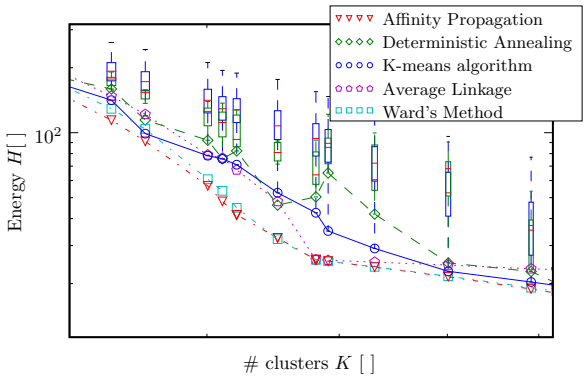
(d)

Fig. 19.3: Without noise. Example On Grid in 2D - Different Clustersizes (uniformly drawn from 0 to 60) - Same Variances. $D = 2$, $K = 49$, $N = 1470$, $N_{\text{noise}} = 500$. K-means is restarted 400 times, DA 20 times. In (b) one can see, that if there is noise, then Ward's method suddenly fails to find good solutions. Here, AP clearly outperforms the competing methods. (c) AP's internal cluster validation is quite inconclusive. With a little imagination, one can still see the plateau at ≈ 50.

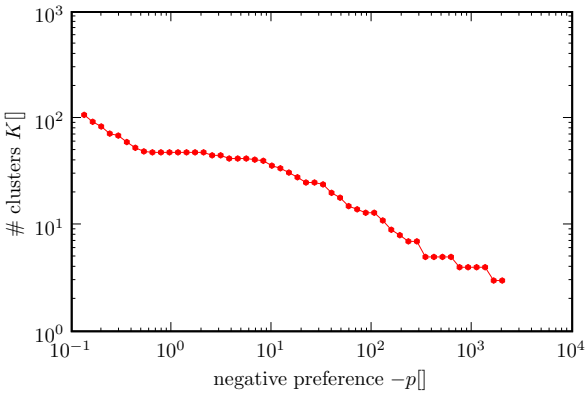
19.3.2 Without Noise



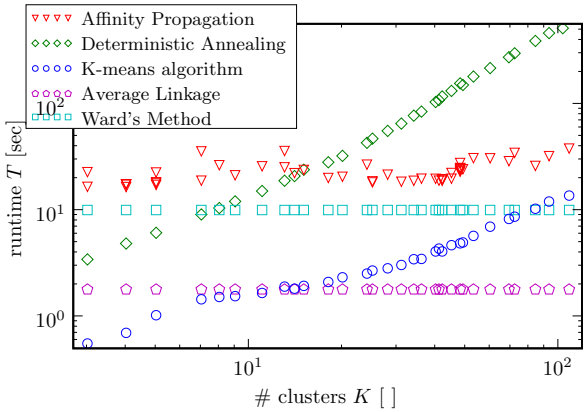
(a) Plot over the whole range of K.



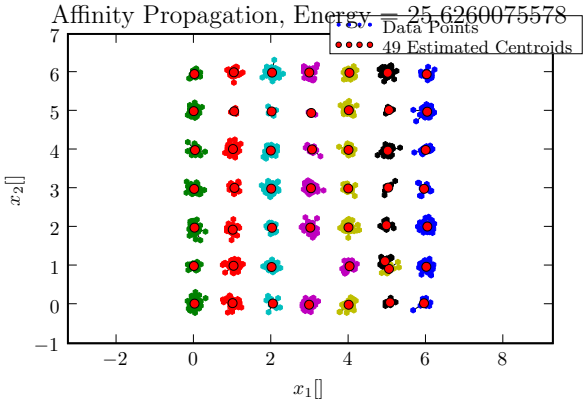
(b) Zoomed to the area of interest around $K = 49$.



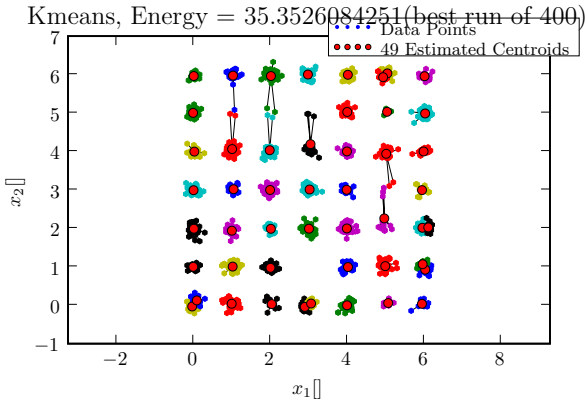
(c) There is a plateau at $K \approx 50$.



(d)



(e)

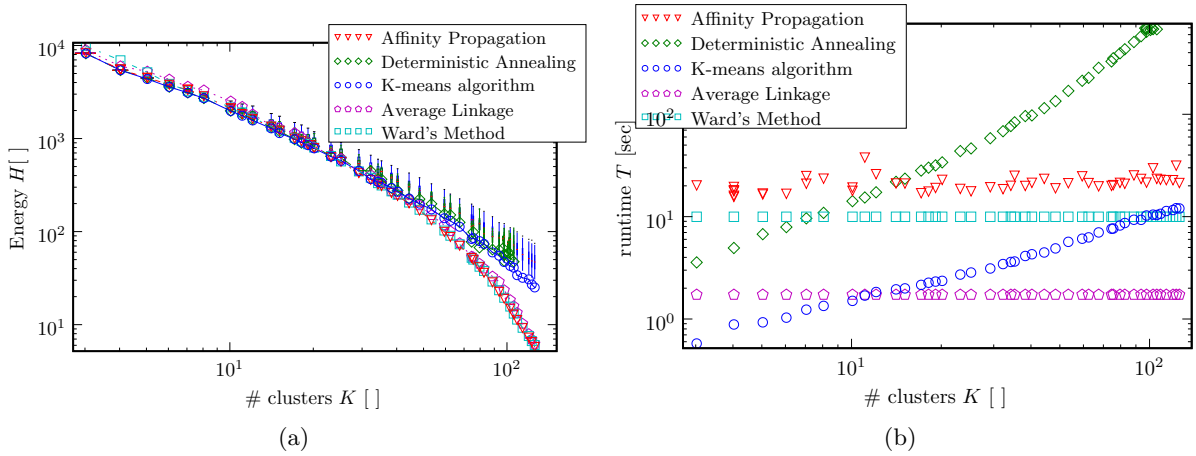


(f)

Fig. 19.4: Without noise. The clustering solution found for $K = 49$. In (a) and (b) one can see that AP and Ward's method outperform the other algorithms. (c) shows that the cluster validation is not as clear as with fixed cluster size. (d) shows the runtimes. AP is approximately constant. K-Means and DA grow approximately linearly. (e) shows the solution found by AP and (f) shows the best solution of 400 restarts found by the K-means algorithm. Apparently, 400 restarts is not enough to find the global optimum. The cluster sizes are variable: each cluster contains 0 to 20 datapoints (uniformly drawn).

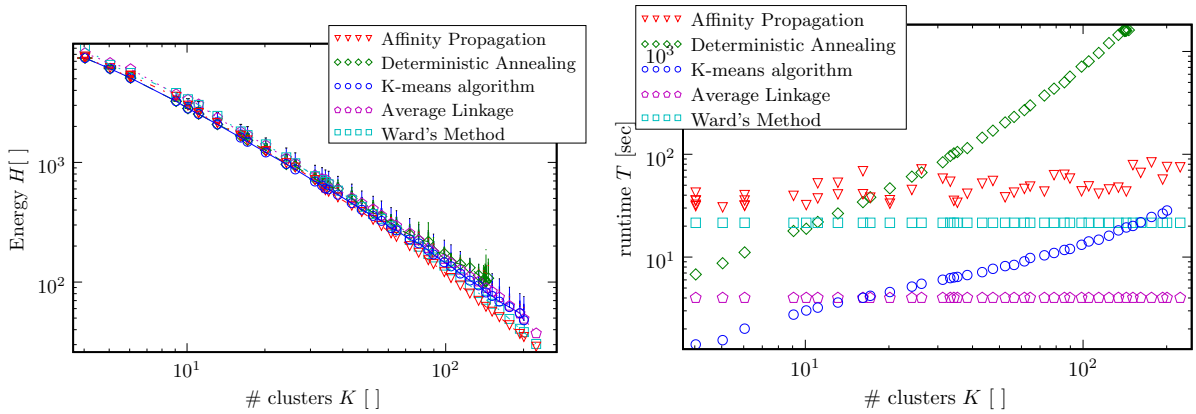
19.4 Random Positions in 2D

We were curious, how the algorithms behave when the positions of the clusters are random The toy dataset we have used was generated in the following way: all clusters have the same variance but different number of data points ranging from 0 to 20. The dataset is generated as explained in 19.1 `# RANDOM WELL DEFINED CLUSTERS IN SMALL DIMENSIONS WITH EQUAL VARIANCES`.



(a) (b)

Fig. 19.5: Without noise. One can see that Ward's method and AP clearly find a better solution that the K-means algorithm and DA.



(a) (b)

Fig. 19.6: With additional 500 noisy data points. The difference between the algorithms shrinks, when there is noise between the clusters. AP finds a little better solution than Ward's method.

19.5 Random Positions, Random Cluster Sizes, Random Variances in 2D

We are interested in the performance of the algorithms when the clusters have random locations. The dataset is generated as explained in 19.1 `# RANDOM WELL DEFINED CLUSTERS IN SMALL DIMENSIONS WITH DIFFERENCENT VARIANCES`.

19.5.1 Without Noise

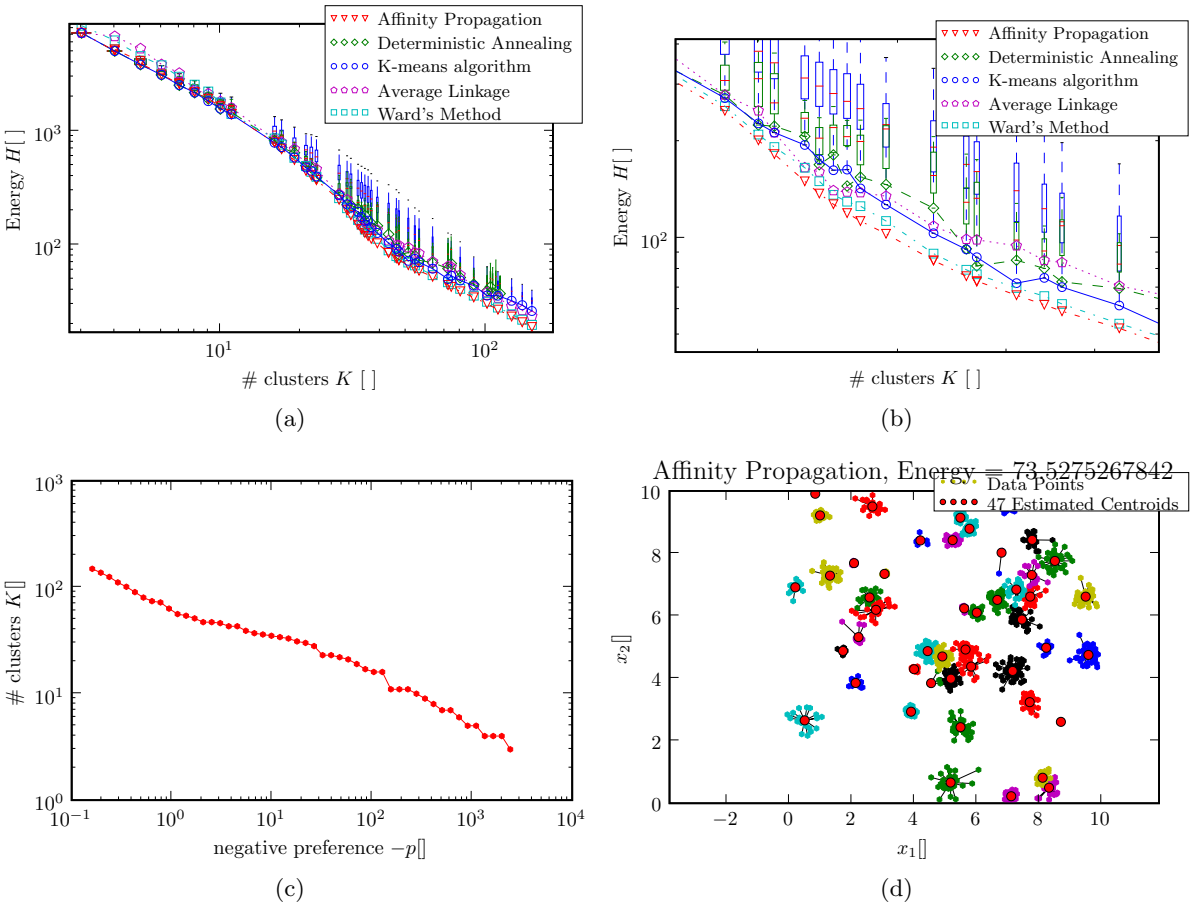


Fig. 19.7: In (a), (b) one can see that Ward's method and AP give much better results than DA and K-means. (c) AP's internal cluster validation is inconclusive since no plateau is visible. (d) shows a typical cluster layout.

19.5.2 With 500 Noisy Data Points

Here, we are interested, how the algorithms perform, when there is a lot of noise. We observe, that Ward's method does not give as nice results anymore, but AP still gives the best answer.

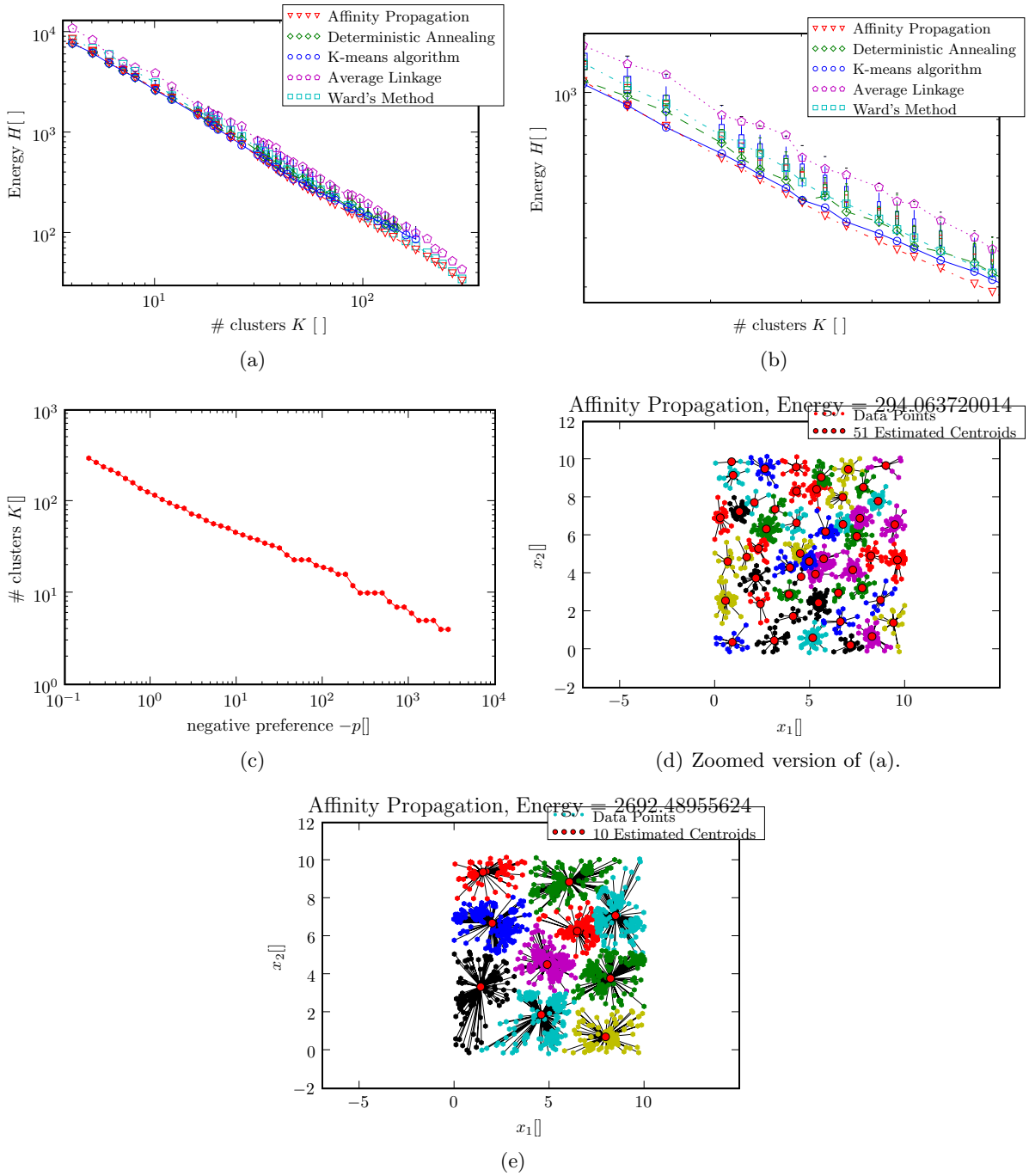
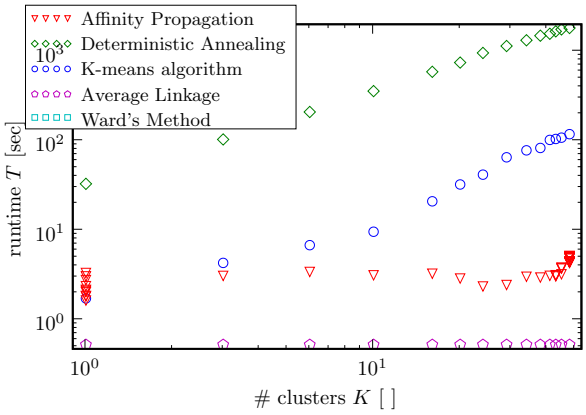


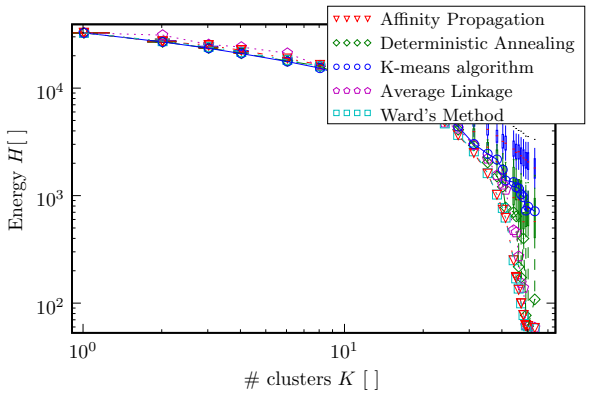
Fig. 19.8: (a) (b) If there are noisy data points, Ward's method performs worse than the other algorithms. In subfigure (e) the cluster layout for $K = 10$ is shown, since subfigure (c) shows a small plateau. Subjectively, this does not seem a good clustering solution. (d) shows the clustering solution at ≈ 50. It is the same dataset as in the case without noise, where the clustering solution of $K = 47$ is depicted. We wondered to what extend the solutions are comparable. As a careful comparison shows, clusters with many data points are also visible when there is a lot of noise.

19.6 Random Positions, Random Cluster Sizes, Random Variances in D=512

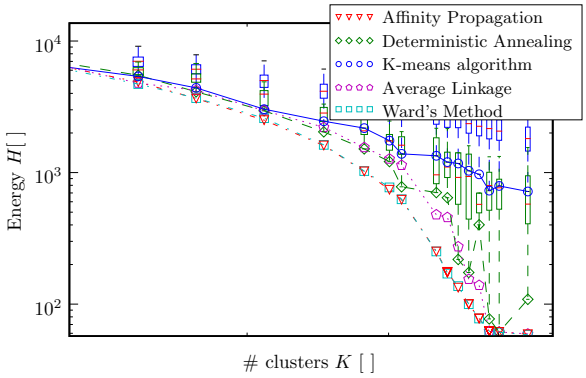
Iterative algorithms that operate on an Euclidean vector space usually have a runtime $\mathcal{O}(D)$ per iteration. Here, we show some examples of toy datasets in high dimensions and with many clusters that are well separated. As one can see in Figure 19.9, AP and Ward's method excel when $K \approx 50$, which is the true number of clusters. The dataset is generated as explained in 19.1 `# WELL DEFINED CLUSTERS GROWING DIMENSIONS`. We picked the dataset with $D = 512$.



(a) Runtime for $D = 512$.



(b) $D = 8$.



(c) Zoomed version of (b).

Fig. 19.9: (a) Even for small K, the K-means algorithm is slower than AP. At $K = 50$, AP is more than one order of magnitude faster than the K-means algorithms and even two orders faster than DA. (b) AP and Ward's method give the best results, followed by DA. The K-means algorithm's result is quite poor.

19.7 Comparison of the Performance when the Dimension Grows

We compare what impact the dimensionality has on the performance of the algorithms. For the dimensions $D = \{2, 4, 8, \ldots, 512\}$ we generated a datasets with $K = 50$ clusters, varying number of elements in the clusters and varying variance. The results are depicted in Figure 19.10. As in the previous subsection, the dataset is generated as explained in 19.1 `# WELL DEFINED CLUSTERS GROWING DIMENSIONS`.

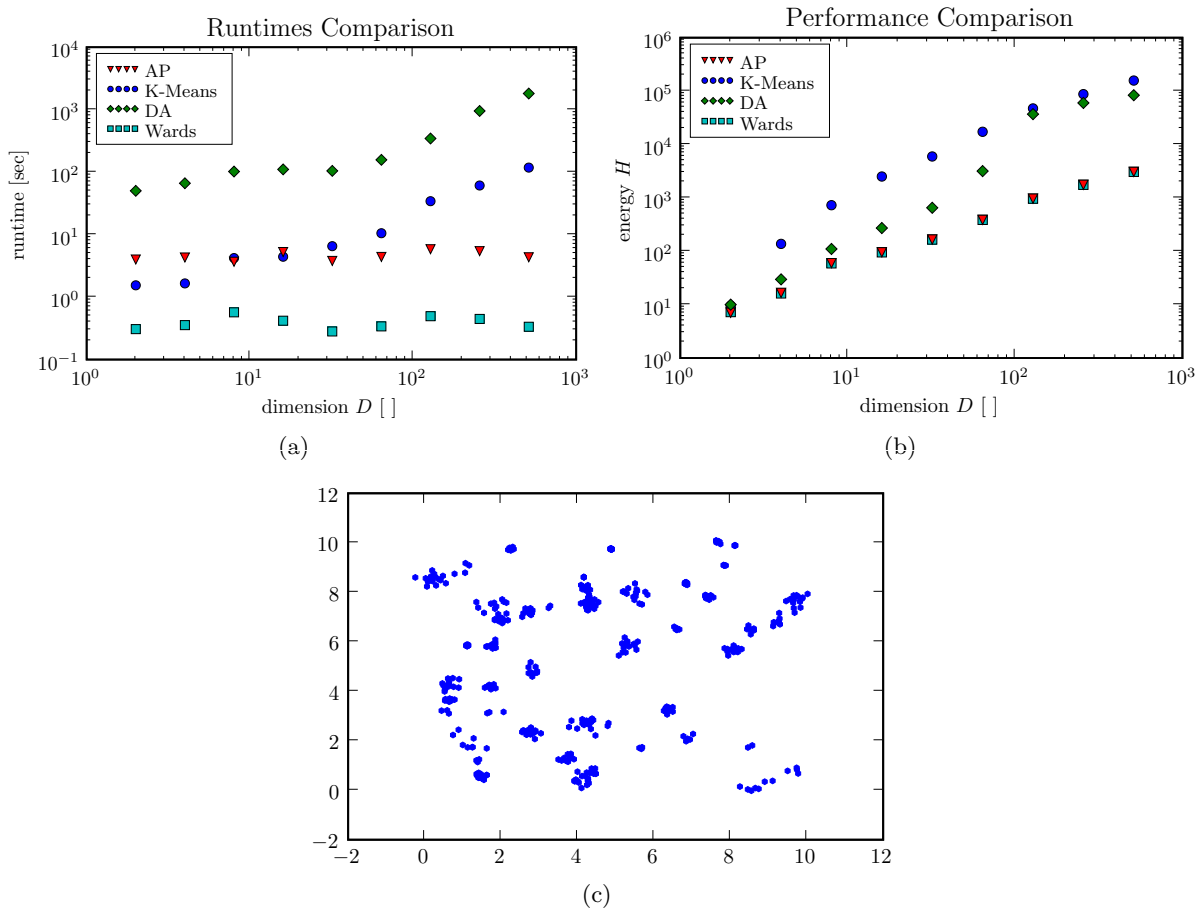


Fig. 19.10: (a) As expected, the runtime for iterative algorithms that operate directly on the vector space with dimension D the runtime grows approximately linearly with D. For high dimensions, AP and Ward's method are much faster than K-Means algorithm or DA. In (b) a plot that indicates the performance of the algorithm is shown. At each dimension, another dataset has been generated with $K = 50$ clusters. We plot the first cluster solution with $K < 50$ that is stable (clusters can overlap, and therefore the effective number of clusters may be smaller). One can see, that DA performs better than the K-Means algorithm for most datasets; however, not as good as AP or Ward's method. (c) Shows a typical generated dataset in $D = 2$.

A Objective Functions

A.0.1 Relation between Mixture of Gaussians / Deterministic Annealing and K-Means Problem

The maximum a posterior objective function of the *model based problem* is given by

$$\theta^*_{\mathrm{MAP}} := \mathrm{argmax}_\theta p(x|\theta)p(\theta) \;, \tag{A.1}$$

where $p(x|\theta)$ is the probability to draw the sample x, i.e. to have the outcome $x = X$ from the random variable X if the underlying parameters are θ. $p(x|\theta)$ is also known as *likelihood function* L. $p(\theta)$ is the a priori knowledge of the distribution of θ, i.e. θ is treated as random variable.

If the *priors* $p(\theta)$ are unknown and assumed to be identical then the MAP problem simplifies to the *maximum likelihood* (ML) problem with objective function

$$\theta^*_{\mathrm{ML}} := \mathrm{argmax}_\theta p(x|\theta) = \mathrm{argmax}_\theta \log p(x|\theta) \;. \tag{A.2}$$

For the special case of a *parametric mixture model* the probability distribution $p(x|\theta)$ is a convex combination of other probability distributions of the same probability distribution only differing in their parameters, i.e.

$$p(x|\theta) \overset{*}{=} \prod_{n=1}^{N} p(x_n|\theta) \overset{\dagger}{=} \prod_{n=1}^{N}\prod_{k=1}^{K} \alpha_k p(x_n|\theta_k) \;, \tag{A.3}$$

where in $*$ we assumed that the data points are *identical, independently distributed* sampled and in † that the probability distribution where the random variable X has been sampled from is a parametric mixture model. In the special case of a *Gaussian mixture model* $p(x_n|\theta_k)$ is given by

$$p(x_n|\theta_k) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}\langle x_n - y_k|\Sigma_k^{-1}|x_n - y_k\rangle} \;, \tag{A.4}$$

where Σ_k is the *covariance matrix* of the k'th cluster. If $\Sigma = diag(\sigma, \ldots, \sigma))$ we write

$$p(x_n|\theta_k) = \frac{1}{(2\pi\sigma^2)^{d/2}} e^{-\frac{1}{2\sigma^2}\langle x_n - y_k|x_n - y_k\rangle} \;. \tag{A.5}$$

The corresponding posterior $p(\theta_k|x_n)$ can be calculated by the use of Bayes formula: In the limes $\sigma \to 0$ the posterior reads

$$\lim_{\sigma\to 0} p(\theta_k|x_n) \;=\; \lim_{\sigma\to 0} \frac{p(x_n|\theta_k)}{\sum_{l=1}^{K} p(x_n|\theta_l)} = M_{kn} \;.$$

B Affinity Propagation

Definition B.1. A *commutative semiring* is a set $\mathbb{Y}$ equipped with two binary operations $\oplus : \mathbb{Y} \times \mathbb{Y} \to \mathbb{Y}$ and $\odot : \mathbb{Y} \times \mathbb{Y} \to \mathbb{Y}$ such that

1. $(\mathbb{Y}, \oplus)$ is a commutative monoid with identity element $\bar{0}$, so that $\forall x, y, z \in \mathbb{Y}$, the following axioms hold:

$$\begin{aligned} (x \oplus y) \oplus z &= x \oplus (y \oplus z) \\ x \oplus y &= y \oplus x \\ \bar{0} \oplus x &= x \end{aligned}$$

2. $(\mathbb{Y}, \odot)$ is a commutative monoid with identity element $\bar{1}$, so that $\forall x, y, z \in \mathbb{Y}$, the following axioms hold:

$$\begin{aligned} (x \odot y) \odot z &= x \odot (y \odot z) \\ x \odot y &= y \odot x \\ \bar{1} \odot x &= x \end{aligned}$$

3. Multiplication distributes over addition:

$$x \odot (y \oplus z) = (x \odot y) \oplus (x \odot z)$$

4. $\bar{0}$ annihilates $\mathbb{Y}$:

$$\bar{0} \cdot x = x \cdot \bar{0} = \bar{0}$$

Note that we the closure $x \oplus y \in \mathbb{Y}$ and $x \odot y \in \mathbb{Y}$ is implicitly guaranteed by the binary operation $\oplus$ and $\odot$. Also, the symbol $\oplus$ or $+$ is usually assumed to be commutative while that is not true for $\odot$ resp. $\cdot$. The word commutative in "commutative semiring" therefore refers to the multiplication $\odot$.

Definition B.2. A *monoid* is a set with a binary operation $\odot : \mathbb{Y} \times \mathbb{Y} \to \mathbb{Y}$ obeying $\forall x, y, z \in \mathbb{Y}$:

1. Associativity $(x \odot y) \odot z = x \odot (y \odot z)$.
2. There exists an identity element $\bar{1} \in \mathbb{Y}$ such that $x \odot \bar{1} = \bar{1} \odot x = x$.

Definition B.3. A *commutative monoid* is a monoid with the additional axiom
Commutivity: $x \odot y = y \odot x$

Definition B.4. A *ring* is a commutative monoid with inverses, i.e. $\forall x \in \mathbb{Y}$ there exists a y, such that $x \odot y = y \odot x = \bar{1}$. The notation for the inverse is either x^{-1} or $-x$.

References

[AM00] S. M. Aji and R. J. McEliece. The Generalized Distributive Law. *IEEE Transactions on Information Theory*, 46(2):325–343, March 2000.

[BWD96] M. Blatt, S. Wiseman, and E. Domany. Superparamagnetic Clustering of Data. *Physical Review Letters*, 76(18):3251–3254, April 1996.

[Cla06] D. P. Clark. *Molecular Biology.* Spektrum - Akademischer Verlag, 2006.

[CR05] N.A. Campbell and J.B. Reece. *Biology - Seventh Edition.* Pearson Education Inc., 2005.

[DFK+99] Drineas, Frieze, Kannan, Vempala, and Vinay. Clustering in large graphs and matrices. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1999.

[DGK07] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors: A multilevel approach. to appear in IEEE Transactions on Pattern Analysis and Machine Intelligence, 2007.

[FD01] J. Fridlyand and S. Dudoit. Applications of resampling methods to estimate the number of clusters and improve the accuracy of a clustering method. September 2001.

[FD05] J. Frey, B. and D. Dueck. Mixture Modeling by Affinity Propagation. 2005.

[FD07a] B. J. Frey and D. Dueck. Clustering by Passing Messages Between Data Points. *Science*, 315:972–977, February 2007.

[FD07b] B. J. Frey and D. Dueck. Supporting Online Material for "Clusting by Passing Messages Between Data Points". *Science*, January 2007.

[Fis06] B. Fischer. *"Complex" Statistical Clustering Models in Image Analysis and Proteomics*. PhD thesis, ETH Zurich, 2006.

[FM83] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78:553–584, 1983.

[FMM+05] B. J. Frey, N. Mohammad, Q. D. Morris, W. Zhang, M. Robinson, S. Mnaimneh, R. Chang, Q. Pan, E. Sat, J. Rossant, B. Bruneau, J. Aubin, B. J. Blencowe, and T. R. Hughes. Genome-wide analysis of mouse transcripts using exon microarrays and factor graphs. *Nature Genetics*, 37:991–996, August 2005.

[HB97] T. Hofmann and J. M. Buhmann. Pairwise Data Clutering by Deterministic Annealing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(1):1–10, January 1997.

[JD88] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[KFL01] F. R. Kschischang, B. J. Frey, and Loeliger. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory*, 47(2):498–519, February 2001.

[Kuh55] H. Kuhn. The Hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[LRBB04] T. Lange, V. Roth, M. Braun, and J.M. Buhmann. Stability-Bases Validation of Clustering Solutions. *Neural Computation*, 16:1299–1323, 2004.

[Mac99] MacKay. Good error-correcting codes based on very sparse matrices. *IEEETIT: IEEE Transactions on Information Theory*, 45, 1999.

[Pen07] E. Pennisi. Working the (Gene Count) Numbers: Finally, a Firm Answer? *Science*, 316:1113, May 2007.

[PHB99] Jan Puzicha, Thomas Hofmann, and Joachim M. Buhmann. Histogram clustering for unsupervised segmentation and image retrieval. *Pattern Recognition Letters*, 20(9):899–909, 1999.

[Ros98] K. Rose. Deterministic Annealing for Clustering, Compression, Classiffication, Regresseion and Related Optimization Problems. *Proceedings of the IEEE*, 86(11):2210–2239, November 1998.

[Sci] Scipy, Package for Scientific Computations in Python. http://www.scipy.org/.

[SM00] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[Thea] The Boost C++-Libraries. http://www.boost.org.

[Theb] The Boost Graph Library. http://www.boost.org/libs/graph/doc/index.html.

[Thec] The Boost Multi Array Library. http://bonsai.ims.u-tokyo.ac.jp/ mdehoon/software/cluster/so

[Thed] The Boost Multi Array Library. http://boost.org/libs/multi_array/doc/index.html.

[Thee] The NumUtil Script, http://www.eos.ubc.ca/research/clouds/software.html. Austin, p.