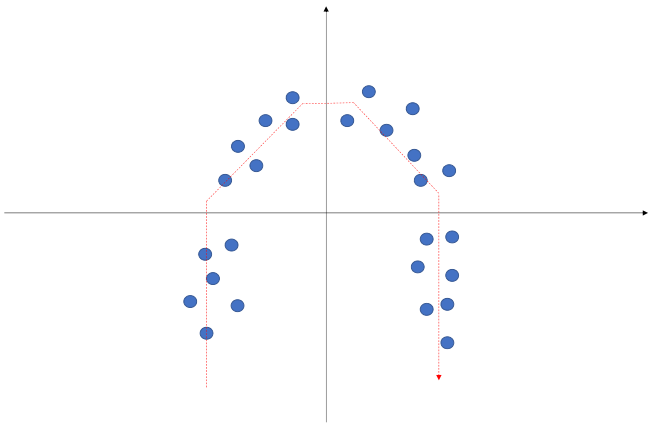
Intelligent Data Analysis
Lecture Notes on
Clustering, Topographic Maps

Peter Tiňo

So far we know how to reduce the dimensionality of large complex data sets and perform queries for terms in a library containing many documents. Of course, one has to be careful - PCA is a *linear* dimensionality reduction technique. Consider, for example, the scenario where we plot some two-dimensional data on a graph and we obtain the following result:

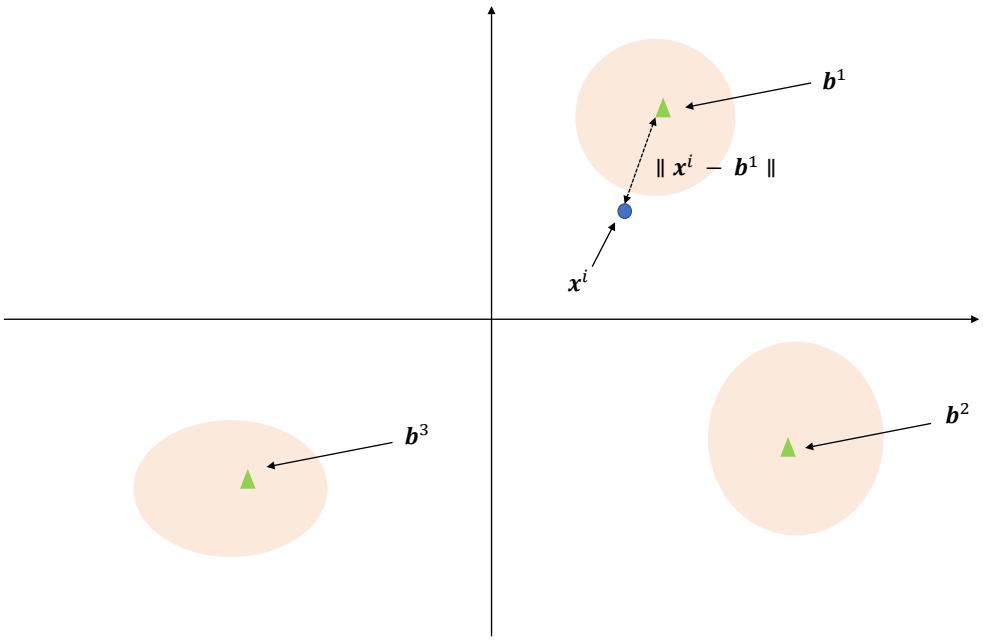


We see that the data is inherently one-dimensional but does not lie along any linear space. PCA does not provide us with a way to find non-linear axes!

For our next topic, we would like to see if there are any natural "groups" or "clusters" within our data. This is a different question about the data that will require a different technique. Such a question can be motivated e.g. in the lossy communication setting: Suppose we want to communicate our data set $\mathcal{D}$ through some sort of information channel. If $|\mathcal{D}|$ (the size of $\mathcal{D}$) and the data dimensionality d is large, transmitting the full data will be very expensive. We want to find a way to compress the data whilst minimizing the information loss when doing so.

1 Vector Quantization

Let $\mathcal{D} = \{\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N\}$ be our data set, $\mathbf{x}^i \in \mathbb{R}^d$. $\mathcal{D}$ has $N \cdot d$ pieces of information (numbers), and our aim is to reduce the information we need to transmit about $\mathcal{D}$. What we can do is to represent our points using a small number of vectors $\mathcal{B} = \{\mathbf{b}^1, \mathbf{b}^2, \ldots, \mathbf{b}^M\}$, $\mathbf{b}^j \in \mathbb{R}^d$, with $M << N$, and transmit those instead. Each of these $\mathbf{b}^j$ are called *codebook vectors*. The set $\mathcal{B}$ is called the *codebook*.



Instead of transmitting a data point $\mathbf{x}^i$, we only need to transmit the codebook vector closest to it. To find which codebook vector is closest to $\mathbf{x}^i$, for each $\mathbf{b}^j$ we need to calculate the distance $||\mathbf{x}^i - \mathbf{b}^j||$ and choose the index of that codebook vector for which the distance is minimal. This index can be interpreted as the index of the *winner* codebook vector in the competition (among the codebook vectors) for the 'right' to represent $\mathbf{x}^i$:

$$win(i) = \arg \min_{j=1,2,\ldots,M} ||\mathbf{x}^i - \mathbf{b}^j||.$$

Assuming that we know $win(i)$ for each $\mathbf{x}^i$, we now only need to transmit $Md+N << Nd$ pieces of information; we first transmit our M codebook vectors, and for each data point $\mathbf{x}^i$, $i = 1, 2, ...N$, we only send the codebook index $win(i)$. Of course this makes sense only if the data is organized in relatively tight groups.

1.1 Error Analysis: Placing the Codebook Vectors

To have our M codebook vectors accurately represent all the points in our data set, we need to minimize the *quantization error* $E(M)$, given by

$$E(M) = \frac{1}{N}\sum_{i=1}^{N} ||\mathbf{x}^i - \mathbf{b}^{win(i)}||.$$

Note that some prefer to omit the normalization factor $\frac{1}{N}$, but for our purposes this does not matter. We can employ the following algorithm to 'intelligently' place the codebook vectors in $\mathbb{R}^d$ such that we minimize $E(M)$:

1) Randomly place codebook vectors $\mathbf{b}^1, \ldots, \mathbf{b}^M$ in $\mathbb{R}^d$. They can be initially placed where some $\mathbf{x}^i$ are positioned.

2) For each $\mathbf{x}^i$ do:

   a) Find the closest codebook vector $\mathbf{b}^{win(i)}$,

   b) Move $\mathbf{b}^{win(i)}$ a bit closer to $\mathbf{x}^i$ by computing $\mathbf{b}^{win(i)}_{new}$:

$$\mathbf{b}^{win(i)}_{new} = \mathbf{b}^{win(i)}_{old} + \eta(\mathbf{x}^i - \mathbf{b}^{win(i)}_{old}).$$

   $\eta > 0$ is known as the *learning rate*.

Note that if $\eta = 1$ in (1), the codebook vector will immediately be placed at $\mathbf{x}^i$, thereby eliminating any quantization error between $\mathbf{b}^{win(i)}$ and $\mathbf{x}^i$. However, it is a greedy approach because this will increase the error between the codebook vector and the neighboring data points clustered with $\mathbf{x}^i$. Remember that all of these points will have $\mathbf{b}^{win(i)}$ as their representative. Hence, η should be smaller than 1. In addition, for a constant value of η, the codebook vectors will never settle down to a single position, even when approaching the cluster means. They will be always chasing the training points $\mathbf{x}^i$ as they are picked in the update process above. Therefore, η needs to gradually decrease over time.

2 The Learning Rate

Let $t \geq 0$ denote the 'time step' when using our algorithm. So $\mathbf{b}^j(t)$ represents the position of the codebook vector $\mathbf{b}^j$ at time t. To decrease the learning rate over time we can suggest an exponential decay in η,

$$\eta(t) = \eta(0) \cdot e^{-\frac{t}{\tau}},$$

where $\eta(0)$ is the initial learning rate. The parameter $\tau > 0$ can be thought of as the 'time scale' of the algorithm i.e. how fast the learning rate decreases. For example, if τ is equal to 60 and time ticks in minutes, then the value of η decreases at an hourly rate. Larger values of τ will result in slower decrease of the learning rate. How τ is determined varies from situation to situation, but finding a reasonable value involves running the algorithm several times and see how small τ can be without hurting the solution (increasing the quantization error).

We now give an iterative version of our algorithm as a function of t:

1) Let $\mathbf{b}^j(0)$ be the position of codebook vector $\mathbf{b}^j$ at $t = 0$. They can be initially placed at randomly selected data points in the data set.

2) For each $\mathbf{x}^i$ do:

   a) Find the closest codebook vector $\mathbf{b}^{win(i)}(t)$,

   b) Move $\mathbf{b}^{win(i)}(t)$ a bit closer to $\mathbf{x}^i$ by computing $\mathbf{b}^{win(i)}(t+1)$:

$$\mathbf{b}^{win(i)}(t+1) = \mathbf{b}^{win(i)}(t) + \eta(t)(\mathbf{x}^i - \mathbf{b}^{win(i)}(t)). \quad (1)$$

Since our algorithm is randomly initialized, we would need to run it several times, each time with a different initialization of the codebook vectors, and then pick the solution that has the smallest quantization error.

3 How many codebook vectors do we need?

The next problem we need to address is setting the parameter M - the number of codebook vectors M we need to represent the data (or, equivalently, the number of 'natural' clusters we think the data contains). Obviously, if we set $M = N$, the quantization error $E(M)$ would be equal to zero (we simply put a codebook vector on top of every point $\mathbf{x}^i$). But this is not a very exciting solution - remember that we are looking for natural groupings in the data.

Figure 1 shows a data set grouped into three well-separated clusters, where we optimally (with respect to the overall quantization error) place either one, two or three codebook vectors. Notice that having three codebook vectors is optimal in our scenario. If we place 4 codebook vectors, $E(M)$ would still decrease, but at a slower rate, since we begin to internally split individual clusters.

To find the optimal number of codebook vectors, we have to investigate the quantization error $E(M)$ as M varies when we employ the algorithm. The error $E(M)$ decreases with increasing M, but the rate of this decrease can change. We need to identify the "knee" $(M_*, E(M_*))$ in the plot $(M, E(M))$, where the rate of decrease in $E(M)$ changes from faster to slower. We would then take M_* as an indication of the 'optimal' number of codebook vectors.

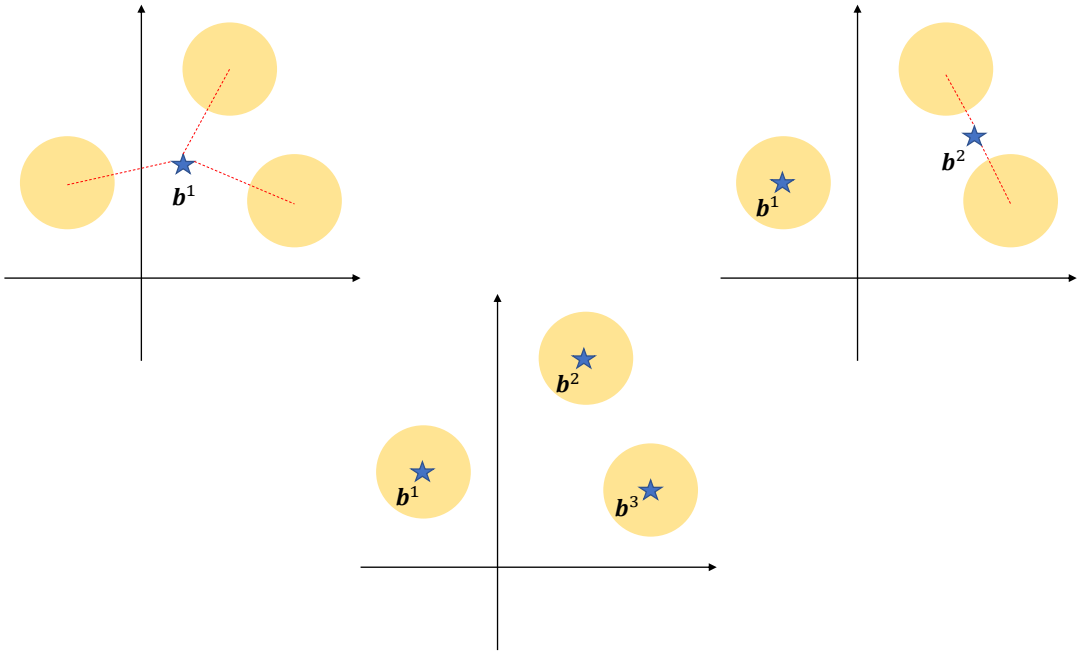
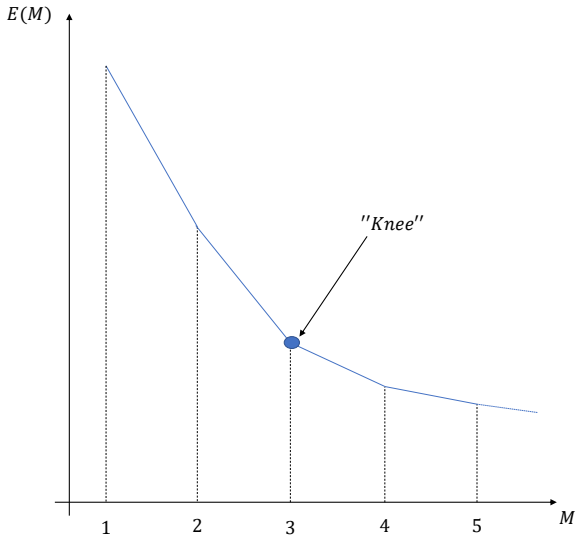


Figure 1: How the codebook vectors are placed for $M = 1, 2$ and 3. Note that the quantization error always decreases as M increases.



4 Topographic Mapping

We now return to the problem of finding curvilinear system of axis in cases the data is inherently low dimensional, embedded in a higher dimensional space, but its low-dimensional structure is *non-linear*. Consider, for example, the set of 3-dim points shown in figure 2.

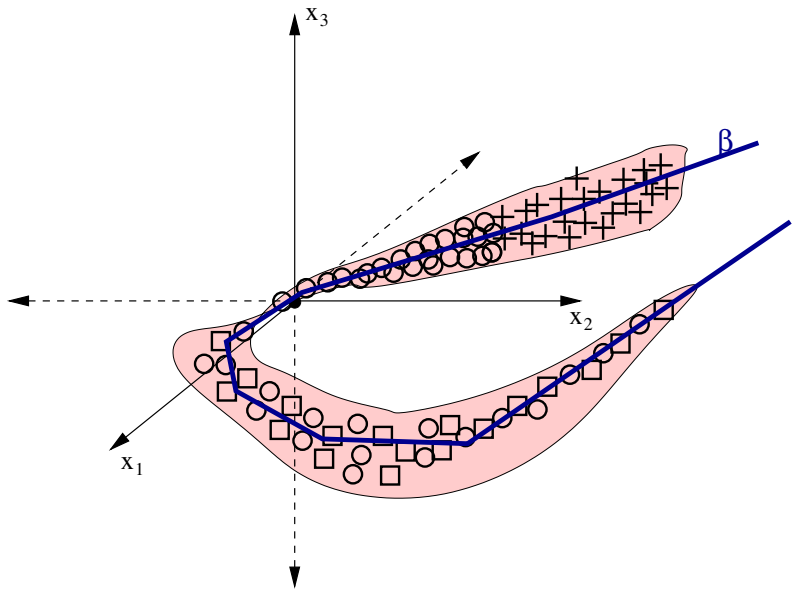


Figure 2: Points organized along a 1-dimensional manifold.

In such cases we can discover the hidden 1-dimensional structure of high-dimensional points by running an *online vector quantization* (VQ) on them, but *under the constraint that the codebook vectors must lie on a one-dimensional "bicycle chain"*. This will cause the codebook vectors to be organized along the "curved noisy tube' (nonlinear 1-dimensional manifold), as illustrated in figure 3. But how can we achieve this when updating the codebook vectors in an on-line vector quantization method described above? One simple idea is to prescribe an *abstract neighborhood structure of codebook vectors* before any training begins. The 1-dimensional bicycle chain structure in figure 3 is an example of such a neighborhood structure. Then, during the codebook update process, for each point $\mathbf{x}^i$, instead of updating just the winner codebook $\mathbf{b}^{win(i)}$, we *update all codebook vectors, but not all to the same degree*. The closer is the codebook vector in the abstract neighborhood structure to $\mathbf{b}^{win(i)}$, the more it gets updated (pushed towards $\mathbf{x}^i$). A codebook vector far away in the abstract neighborhood structure from $\mathbf{b}^{win(i)}$ (e.g. on the other side of the bicycle chain from $\mathbf{b}^{win(i)}$) will get updated only negligibly.

Formally, we can represent the degree to which a codebook vector $\mathbf{b}^j$ is prescribed to be a neighbor of $\mathbf{b}^{win(i)}$ through a *neighborhood function* $h(win(i), j)$ taking values in the interval $[0, 1]$. We stress that the prescribed neighborhood relations between codebook vectors (which codebook vectors should be placed next to each other in the data space) are decided before seeing the actual data. Hence, the neighborhood function operates only on the codebook vector indices, not on their actual positions in the data space! Moreover, we would, of course, like the value of the neighborhood function to be maximized if $j = win(i)$, i.e. $h(win(i), win(i)) = 1$. We would also like the value of $h(win(i), j)$ to diminish the further away the index j is in the prescribed neighborhood structure from $win(i)$.

One widely used formulation of the neighborhood function is through a Gaussian kernel

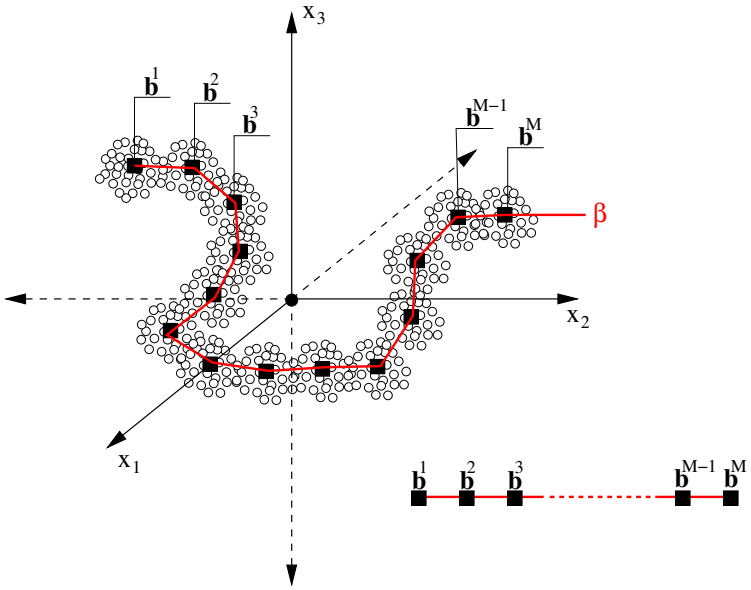


Figure 3: 1-dimensional "bicycle chain" organization of codebook vectors.

positioned on top of $win(i)$:

$$h(win(i), j) = e^{-\frac{(win(i)-j)^2}{\sigma^2}}.$$

Here, $(win(i) - j)^2$ is the (square) distance between the indexes $win(i)$ and j on the bicycle chain and σ is the parameter controlling the 'width' of the neighborhood.

Initially, the neighborhood width σ should be large in order to allow for broad cooperation of codebook vectors, so that all of them get into the cloud of points. In this stage the model gets roughly organized within the data points. Once that has been achieved, a more refined fine-tuning of the codebook vector positions can start. This can be achieved by allowing the neighborhood width to get more and more specific (smaller) so that the order of indexes as prescribed in the bicycle chain is strictly imposed.

Analogously to the learning rate, we can suggest an exponential decay in σ:

$$\sigma(t) = \sigma(0) \cdot e^{-\frac{t}{\nu}},$$

where $\nu > 0$ is a 'time scale' controlling how fast the neighborhood width decays and $\sigma(0)$ is the initial neighborhood width.

The modified algorithm now reads:

1) Let $\mathbf{b}^j(0)$, $j = 1, 2, ..., M$, be the randomized initial positions of codebook vectors $\mathbf{b}^j$ at $t = 0$.

2) For each $\mathbf{x}^i$ do:

   a) Find the closest codebook vector $\mathbf{b}^{win(i)}(t)$,

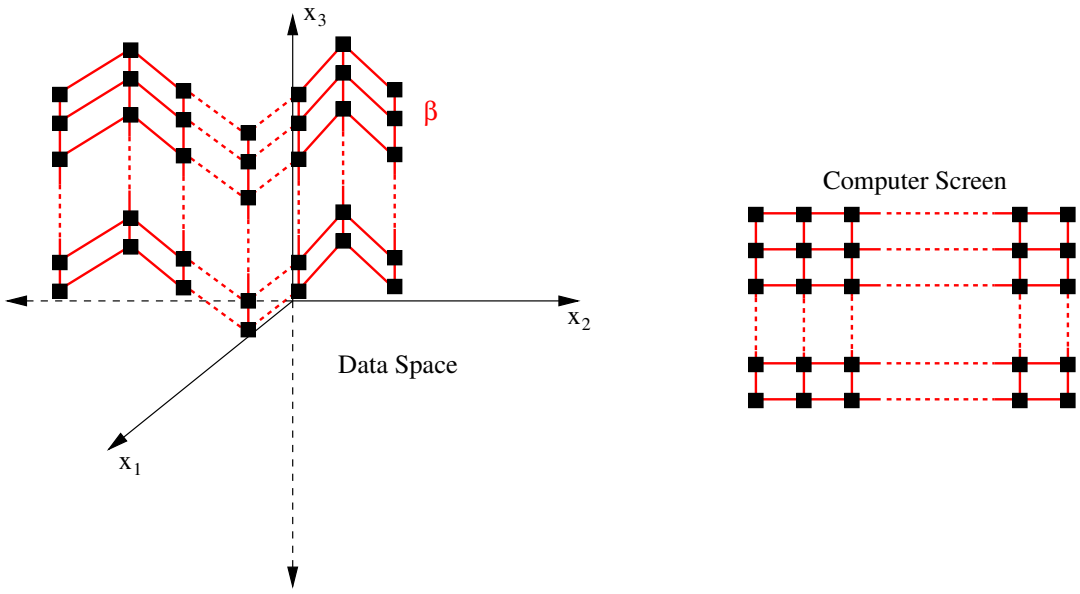


Figure 4: Generalization of the 1-dimensional neighborhood structure of the bicycle chain to a 2-dimensional structure of the "fishing net".

b) Move all codebook vectors a bit closer to $\mathbf{x}^i$ (to the degree prescribed by the learning rate *and the neighborhood function*):

$$\mathbf{b}^j(t+1) = \mathbf{b}^j(t) + \eta(t) \cdot h(win(i), j) \cdot (\mathbf{x}^i - \mathbf{b}^{win(i)}(t)). \tag{2}$$

Of course, we can generalize the neighborhood structure of bicycle chain to any convenient structure, for example a 2-dimensional grid on our computer screen. We can simply cover the computer screen with a regular 2-dimensional grid of nodes. This is illustrated in figure 4.

Finally, once the 2-dimensional topographic mapping is learnt, we can visualize high-dimensional data points $\mathbf{x}^i$ as their 2-dimensional projections on the computer screen. This can be simply done by identifying the projection $\tilde{\mathbf{x}}^i$ with the index (grid point) $win(i)$ on the computer screen corresponding to the codebook vector $\mathbf{w}^{win(i)}$ closest to $\mathbf{x}^i$ in the data space.