Chapter 4

Reducing the computational cost

As emphasized in Section 3.1, Neighbourhood Component Analysis (NCA) is a computationally expensive method. The evaluation of its objective function is quadratic in the size of the data set. Given that the optimization is done iteratively, NCA becomes prohibitively slow when applied on large data sets. There is only little previous work that uses NCA for large scaled applications. One example is Singh-Miller (2010), who parallelizes the computations across multiple computers and adopts various heuristics to prune terms of the objective function and the gradient.

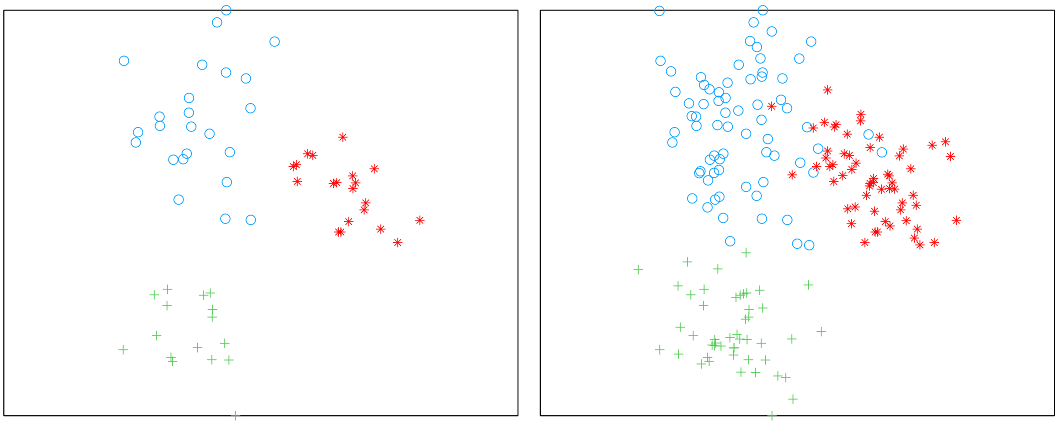
This chapter aims to complete the existing work. We present a series of methods that can improve NCA's speed. Most of these ideas are new in the context of NCA. They are also quite versatile. Each method proposes a new objective function and can be regarded as an independent model on its own.

4.1 Sub-sampling

Sub-sampling is the simplest idea that can help speeding up the computations. For the training procedure we use a randomly selected sub-set $\mathcal{D}_n$ of the original data set $\mathcal{D}$:

$$\mathcal{D}_n = \{\mathbf{x}_{i_1}, \cdots, \mathbf{x}_{i_n}\} \subseteq \mathcal{D}.$$

If n is the size of the sub-set then the cost of the gradient is reduced to $\mathcal{O}(dDn^2)$. After the projection matrix $\mathbf{A}$ is learnt, we apply it to the whole data set $\{\mathbf{x}_i\}_{i=1}^N$. Then all the new data points $\{\mathbf{A}\mathbf{x}_i\}_{i=1}^N$ are used for classification. The cost of the classification is $\mathcal{O}(dN)$; it is only linear in the total number of points N.



(a) Learnt projection $\mathbf{A}$ on the sub-sampled data set $\mathcal{D}_n$.

(b) The projection $\mathbf{A}$ applied to the whole data set $\mathcal{D}$.

Figure 4.1: Result of sub-sampling method on `wine`. There were used one third of the original data set for training, *i.e.*, $n = N/3$. We note that the points that belong to the sub-set $\mathcal{D}_n$ are perfectly separated. But after applying the metric to the whole data there appear different misclassification errors. The effects are even more acute if we use smaller sub-sets.

While easy to implement, this method discards a lot of information available. Also it is affected by the fact the sub-sampled data has a thinner density than the real data. The distances between the randomly selected points are larger than they are in the full data set. This causes the scale of the projection matrix $\mathbf{A}$ not to be large enough, Figure 4.1.

4.2 Mini-batches

The next obvious idea is to use sub-sets in an iterative manner, similarly to the stochastic gradient descent method: split the data into mini-batches and train on them successively. Again the cost for one evaluation of the gradient will be $\mathcal{O}(dDn^2)$ if the mini-batch consists of n points.

There are different possibilities for splitting the data-set:

1. Random selection. In this case the points are assigned randomly to each mini-batch and after one pass through the whole data set another random allocation is done. As in section 4.1, this suffers from the thin distribution problem. In order to alleviate this and achieve convergence, large-sized

Algorithm 4.1 Training algorithm using mini-batches formed by clustering

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and initial linear transformation $\mathbf{A}$.
1: repeat
2:     Project each data point using $\mathbf{A}$: $\mathcal{D}_\mathbf{A} = \{\mathbf{A}\mathbf{x}_1, \cdots, \mathbf{A}\mathbf{x}_N\}$.
3:     Use either algorithm 4.2 or 4.3 on $\mathcal{D}_\mathbf{A}$ to split $\mathcal{D}$ into $K$ mini-batches $\mathcal{M}_1, \cdots, \mathcal{M}_K$.
4:     for all $\mathcal{M}_i$ do
5:         Update parameter: $\mathbf{A} \leftarrow \mathbf{A} + \eta \frac{\partial f(\mathbf{A}, \mathcal{M}_i)}{\partial \mathbf{A}}$.
6:         Update learning rate $\eta$.
7:     end for
8: until convergence.
```

mini-batches should be used (similar to Laurens van der Maaten's implementation). The algorithm is similar to Algorithm 4.1, but lines 2 and 3 will be replaced with a simple random selection.

2. Clustering. Constructing mini-batches by clustering ensures that the point density in each mini-batch is conserved. In order to maintain a low computational cost, we consider cheap clustering methods, *e.g.*, farthest point clustering (FPC; Gonzalez, 1985) and recursive projection clustering (RPC; Chalupka, 2011).

   FPC gradually selects cluster centres until it reaches the desired number of clusters K. The point which is the farthest away from all the current centres is selected as new centre. The precise algorithm is presented below.

Algorithm 4.2 Farthest point clustering

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and $K$ number of clusters.
Randomly pick a point that will be the first centre $\mathbf{c}_1$.
Allocate all the points in the first cluster $\mathcal{M}_1 \leftarrow \mathcal{D}$.
for $i = 1$ to $K$ do
    Select the $i$-th cluster centre $\mathbf{c}_i$ as the point that is farthest away from any cluster centre $\mathbf{c}_1, \cdots, \mathbf{c}_{i-1}$.
    Move to the cluster $\mathcal{M}_i$ those points that are closer to its centre than to any other cluster centre: $\mathcal{M}_i = \{\mathbf{x} \in \mathcal{D} \mid d(\mathbf{x}; \mathbf{c}_i) < d(\mathbf{x}; \mathbf{c}_j), \forall j \neq i\}$
end for
```

The computational cost of this method is $\mathcal{O}(NK)$. However, we do not have any control on the number of points in each cluster, so we might end up with very unbalanced clusters. A very uneven split has a couple of obvious drawbacks: too large mini-batches will maintain high cost, while on too small clusters there is not too much to learn.

An alternative is RPC which was especially designed to mitigate this problem. It constructs the clusters similarly to how the k-d trees are build, Subsection 4.4.1. However instead of splitting the data set across axis aligned direction it chooses the splitting directions randomly, Algorithm 4.3. So, because it uses the median value it will result in similar sized clusters and we can easily control the dimension of each cluster.

Algorithm 4.3 Recursive projection clustering

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and $n$ size of clusters.
if $N < n$ then
    New cluster: $i \leftarrow i + 1$.
    return current points as a cluster: $\mathcal{M}_i \leftarrow \mathcal{D}$.
else
    Randomly select two points $\mathbf{x}_j$ and $\mathbf{x}_k$ from $\mathcal{D}$.
    Project all data points onto the line defined by $\mathbf{x}_j$ and $\mathbf{x}_k$. (Give equation?)
    Select the median value $\tilde{\mathbf{x}}$ from the projected points.
    Recurs on the data points above and below $\tilde{\mathbf{x}}$: RPC($\mathcal{D}_{>\tilde{\mathbf{x}}}$) and RPC($\mathcal{D}_{\leq\tilde{\mathbf{x}}}$).
end if
```

Note that we are re-clustering in the transformed space after one sweep through the whole data set. There are also other alternatives. For example, we could cluster in the original space. This can be done either only once or periodically. However the proposed variant has the advantage of a good behaviour for a low-rank projection matrix $\mathbf{A}$. Not only that is cheaper, but the clusters resulted in low dimensions by using RPC are closer to the real clusters then applying the same method in a high dimensional space.

4.3 Stochastic learning

The following technique is theoretically justified by stochastic approximation arguments. The main idea is to get an unbiased estimator of the gradient by looking only at a few points and how they relate to the entire data set.

More precisely, in the classical learning setting, we update our parameter $\mathbf{A}$ after we have considered each point $\mathbf{x}_i$ in the data set. In the stochastic learning procedure, we update $\mathbf{A}$ more frequently by considering only n randomly selected points. As in the previous case, we still need to compute the soft assignments $\{p_i\}_{i=1}^{n}$ using *all* the N points. To stress this further, this solution differs from the mini-batch approach. For the previous method, the contributions $\{p_i\}_{i=1}^{n}$ are calculated only between the n points that belong to the mini-batch.

The objective function that we need to optimize at each iteration and its gradient are given by the next equations:

$$f_{\text{sNCA}}(\mathbf{A}) = \sum_{l=1}^{n} p_{i_l} \tag{4.1}$$

$$\frac{\hat{\partial f}}{\partial \mathbf{A}} = \frac{\partial f_{\text{sNCA}}}{\partial \mathbf{A}} = \sum_{l=1}^{n} \frac{\partial p_{l_i}}{\partial \mathbf{A}}, \tag{4.2}$$

$$\text{with } p_i = \sum_{\substack{j=1 \\ j \in c_i}}^{N} p_{ij}. \tag{4.3}$$

This means that the theoretical cost of the stochastic learning method scales with nN.

Algorithm 4.4 Stochastic learning for NCA (sNCA)

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, $n$ number of points to consider for the
    gradient estimation, $\mathbf{A}$ initial linear transformation.
repeat
    Split data $\mathcal{D}$ into groups $\mathcal{M}_i$ of size $n$.
    for all $\mathcal{M}_i$ do
        Update parameter using gradient given by Equation 4.2:
        $\mathbf{A} \leftarrow \mathbf{A} + \eta \frac{\partial f_{\text{sNCA}}(\mathbf{A}, \mathcal{M}_i)}{\partial \mathbf{A}}$.
        Update learning rate $\eta$.
    end for
until convergence.
```

This method comes with an additional facility. It can be used for on-line learning. Given a new point $\mathbf{x}_{N+1}$ we update $\mathbf{A}$ using the derivative $\frac{\partial p_{N+1}}{\partial \mathbf{A}}$.

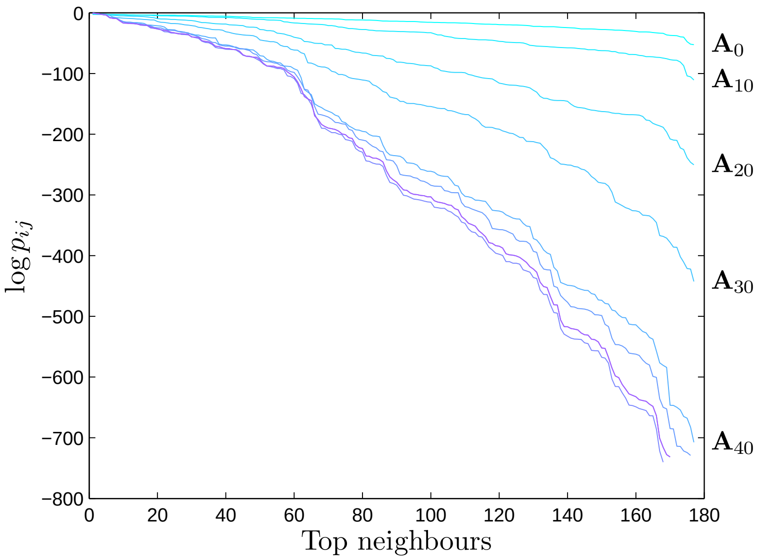


Figure 4.2: Evolution of the stochastic assignments p_{ij} during training for a given point $\mathbf{x}_i$.

4.4 Approximate computations

A straightforward way of speeding up the computations was previously mentioned in the original paper (Goldberger et al., 2004) and in some NCA related work (Weinberger and Tesauro, 2007; Singh-Miller, 2010). The observations involve pruning small terms in the original objective function. We then use an approximated objective function and its corresponding gradient for the optimization process.

The motivation lies in the fact that the contributions p_{ij} decay very quickly with distance:

$$p_{ij} \propto \exp\{-d(\mathbf{A}\mathbf{x}_i; \mathbf{A}\mathbf{x}_j)^2\}.$$

The evolution of the contributions during the training period is depicted in Figure 4.2. We notice that most of the values p_{ij} are insignificant compared to the largest contribution. This suggests that we might be able to preserve the accuracy of our estimations even if we discard a large part of the neighbours.

So, Weinberger and Tesauro (2007) choose to use only the top $m = 1000$ neighbours for each point $\mathbf{x}_i$. Also they disregard those points that are farther away than $d_{\max} = 34$ units from the query point: $p_{ij} = 0, \forall \mathbf{x}_j$ such that $d(\mathbf{A}\mathbf{x}_i; \mathbf{A}\mathbf{x}_j) > d_{\max}$. While useful in practical situations, these suggestions lack of a principled description: how can we optimally choose m and $d_{\max}$ in a general setting? We would also like to be able to estimate the error introduced by the approximations.

We correct those drawbacks by making use of the KDE formulation of NCA (see Section 3.2) and adapting existing ideas for fast KDE (Deng and Moore, 1995; Gray and Moore, 2003) to our particular application. We will use a class of accelerated methods that are based on data partitioning structures (*e.g.*, k-d trees, ball trees). As we shall shortly see, these provide us with means to quickly find only the neighbours $\mathbf{x}_j$ that give significant values p_{ij} for any query point $\mathbf{x}_i$.

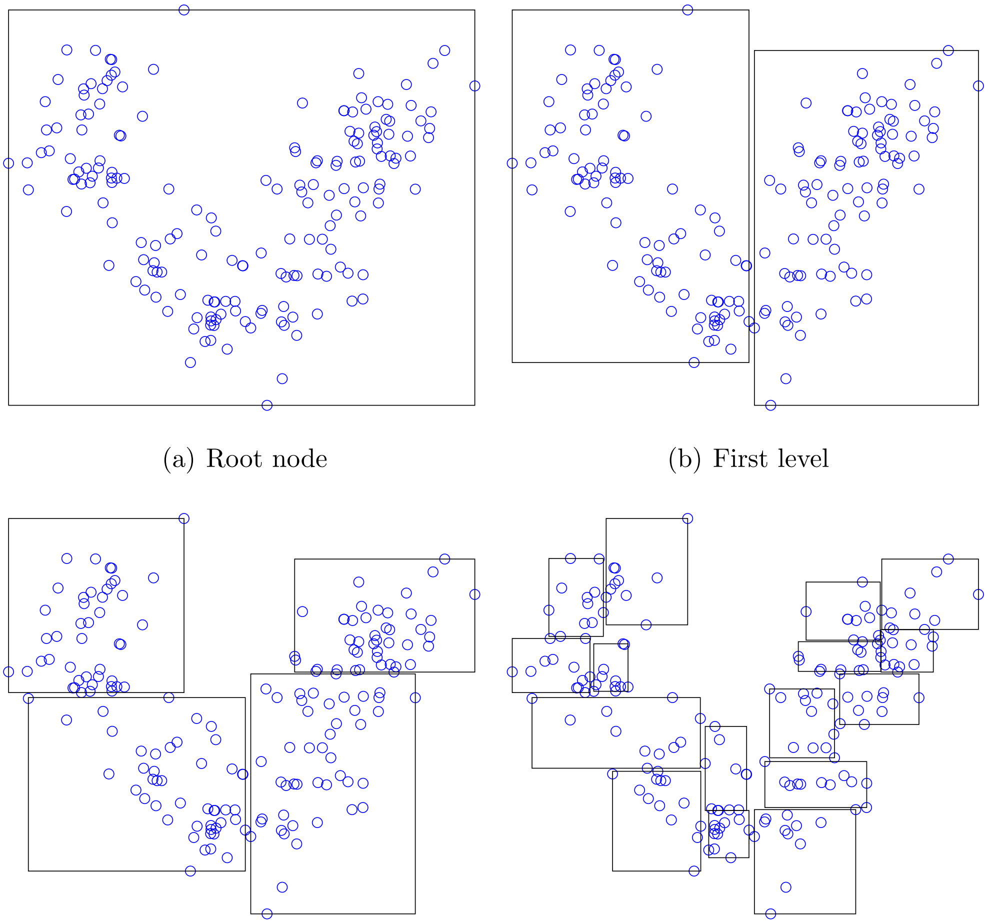
4.4.1 k-d trees

The k dimensional tree structure (k-d tree; Bentley, 1975) organises the data in a binary tree using axis-aligned splitting planes. The k-d tree has the property to place close in the tree those points that live nearby in the original geometrical space. This makes such structures efficient mechanisms for nearest neighbour searches (Friedman et al., 1977) or range searches (Moore, 1991).

There are different flavours of k-d trees. We choose for our application a variant of k-d tree that uses bounding boxes to describe the position of the points. Intuitively, we can imagine each node of the tree as a bounding hyper-rectangle in the D dimensional space of our data. The root node will represent the whole data set and it can be viewed as an hyper-rectangle that contains all the data points, see Figure 4.3(a). In the two-dimensional presented exampled, the points are enclosed by rectangles. From Figure 4.3(a) to 4.3(d), there are presented the existing bounding boxes at different levels of the binary tree. To understand how these are obtained, we discuss the k-d tree construction.

The building of the tree starts from the root node and is done recursively. At each node we select which of the points from the current node will be allocated to each of the two children. Because these are also described by hyper-rectangles, we just have to select a splitting plane. Then the two successors will consist of the points from the two sides of the hyper-plane.

A splitting hyper-plane can be fully defined by two parameters: a direction $\vec{d}$ on which the plane is perpendicular and a point P that is in the plane. Given that the splits are axis aligned, there are D possible directions $\vec{d}$. We can either choose this randomly or we can use each of the directions from 1 to D in a successive manner. A more common approach is to choose $\vec{d}$ to be the dimension



(a) Root node

(b) First level

(c) Second level

(d) Last level

Figure 4.3: Illustration of the k-d tree with bounding boxes at different levels of depths. This figure also outlines the building phases of the tree.

that presents the largest variance:

$$\vec{d} \leftarrow \text{argmax}_d(\max_i x_{id} - \min_i x_{id}). \tag{4.4}$$

This results in a better clustering of the points and the shape of the bounding boxes will be closer to the shape of a square. Otherwise it might happen that points situated in the same node can still be further away.

Regarding the splitting value P, a usual choice is the median value $\tilde{x}_d$ on the previously selected direction $\vec{d}$. This choice of P guarantees a balanced tree which offers several advantages. We can allocate static memory for the entire data structure. This is faster to access than dynamical allocation. Also a balanced tree has a better worst case complexity than an unbalanced one. Other useful implementation tricks that can be applied to balanced k-d trees are suggested by Lang (2009).

After the splitting plane is chosen, the left child will contain the points that are on the left of the hyper-plane:

$$\mathcal{D}_{\leq \tilde{x}_d} = \{\mathbf{x} \in \mathcal{D}_{\mathbf{x}_i} | x_d \leq \tilde{x}_d\}, \tag{4.5}$$

where $\mathcal{D}_{\mathbf{x}_i}$ denotes the data points bounded by the current node $\mathbf{x}_i$. Similarly, the right child will contain the points that are placed on the right of the hyper-plane:

$$\mathcal{D}_{> \tilde{x}_d} = \{\mathbf{x} \in \mathcal{D}_{\mathbf{x}_i} | x_d > \tilde{x}_d\}. \tag{4.6}$$

This process is repeated until the number of points bounded by the current node goes below a threshold m. These nodes are the leaves of the tree and they store the effective points. The other non-leaf nodes store information regarding the bounding box and the splitting plane. Note that a hyper-rectangle is completely defined by only two D-dimensional points, one for the "top-right" corner and the other for the "bottom-left" corner.

The most used operation on a k-d tree is the nearest neighbour (NN) search. While we will not apply pure NN for the next method, we will use similar concepts. However, we can do NN retrieval with k-d trees after we applied NCA, as suggested by Goldberger et al. (2004). The search in the k-d tree is done in a depth-first search manner: start from the root and traverse the whole tree by selecting the closest node to the query point. In the leaf, we find the nearest neighbour from the m points and store it and the corresponding distance $d_{\min}$. Then we recurs up the tree and look at the farther node. If this is situated at

Algorithm 4.5 k-d tree building algorithm

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, $i$ position in tree and $m$ number of points in leaves.
if $N < m$ then
    Mark node $i$ as leaf: splitting_direction(i)=-1.
    Add points to leaf: points(i)$\leftarrow \mathcal{D}$.
    return
end if
Choose direction $\vec{d}$ using Equation 4.4: splitting_direction(i)=$d$.
Find the median value $\tilde{x}_d$ on the direction $\vec{d}$: splitting_value(i)=$\tilde{x}_d$.
Determine the subsets of points that are separated by the resulting splitting plane: $\mathcal{D}_{\leq \tilde{\mathbf{x}}}$ and $\mathcal{D}_{> \tilde{\mathbf{x}}}$, see Equations 4.5 and 4.6.
Build left child build_kdtree($\mathcal{D}_{\leq \tilde{\mathbf{x}}}$,2*i).
Build right child build_kdtree($\mathcal{D}_{> \tilde{\mathbf{x}}}$,2*i+1).
```

a minimum distance that is smaller than d_{min}, we have to investigate also that node. In the other case, we can ignore the node and all the points it contains. Usually, a large fraction of the points can be omitted, especially when the data has structure. It is important to stress that the performance of k-d trees quickly degrades with the number of dimensions the data lives.

4.4.2 Approximate kernel density estimation

The following ideas are mostly inspired by previous work on fast kernel density estimators with k-d trees (Gray and Moore, 2001, 2003; Alexander Gray, 2003) and fast Gaussian Processes regression with k-d trees (Shen et al., 2006).

In kernel density estimation (KDE), we are given a data set $\mathcal{D}$ and the goal is to compute the probability denisty at a given point using a sum of the contributions from all the points:

$$p(\mathbf{x}_i) = \frac{1}{N} \sum_{j=1}^{N} k(\mathbf{x}_i|\mathbf{x}_j). \tag{4.7}$$

A common class of KDE problems are the N-body problems (Gray and Moore, 2001). This imposes to estimate $p(\mathbf{x}_i)$ for each data point $\{\mathbf{x}_i\}_{i=1}^N$ in the data set $\mathcal{D}$. Note that this operation is quadratic in the number of points, as it is the case of NCA.

If the number of samples N in $\mathcal{D}$ is sufficiently large, we expect to accurately approximate $p(\mathbf{x}_i)$ by using only nearby neighbours of $\mathbf{x}_i$.

To illustrate how this works, let us assume we are given a query point $\mathbf{x}_i$ and a group of points G. We try to reduce computations by replacing each individual contribution $k(\mathbf{x}_i|\mathbf{x}_j), \mathbf{x}_j \in G$, with a fixed quantity $k(\mathbf{x}_i|\mathbf{x}_g)$. The value of $k(\mathbf{x}_i|\mathbf{x}_g)$ is group specific and since it is used for all points in G, it is chosen such that it does not introduce a large error. A reasonable value for $k(\mathbf{x}_i|\mathbf{x}_g)$ is obtained by approximating each point $\mathbf{x}_i$ with a fixed $\mathbf{x}_g$, for example, the mean of the points in G. Then we compute the kernel value $k(\mathbf{x}_i|\mathbf{x}_g)$ using the estimated $\mathbf{x}_g$. A second possibility is to directly approximate $k(\mathbf{x}_i|\mathbf{x}_g)$. For example:

$$k(\mathbf{x}_i|\mathbf{x}_g) = \frac{1}{2}\left(\min_j k(\mathbf{x}_i|\mathbf{x}_j) + \max_j k(\mathbf{x}_i|\mathbf{x}_j)\right). \tag{4.8}$$

We prefer the this option, because it does not introduce any further computational expense. Both the minimum and the maximum contributions are previously calculated to decide whether to prune or not. Also it does not need storing any additional statistic, such as the mean.

We see that the error introduced by each approximation is bounded by the following quantity:

$$\epsilon_{\max} \leq \frac{1}{2}\left(\max_j k(\mathbf{x}_i|\mathbf{x}_j) - \min_j k(\mathbf{x}_i|\mathbf{x}_j)\right). \tag{4.9}$$

This can be controlled to be small if we approximate only for those groups that are far away from the query point or when the variation of the kernel value is small within the group. It is better still to consider the error relative to the total quantity we want to estimate. Of course we do not know the total sum we want to estimate in advance, but we can use a lower bound $p_{\mathrm{SoFar}}(\mathbf{x}_i) + \max k(\mathbf{x}_i|\mathbf{x}_j)$. Hence, a possible cut-off rule is:

Note that in this case it is important the order in which we accumulate. A large $p_{\mathrm{SoFar}}(\mathbf{x}_i)$ in the early stage will allow more compuational savings.

We use k-d trees to form groups of points G that will be described as hyper-rectangles. To compute the probability density function $p(\mathbf{x}_i)$, we start at the root, the largest group, and traverse the tree going through the nearest node each time until we reach the leaf. This guarantees that we add large contributions at the begining. Then we recurs up the tree and visit other nodes only if necessary, when the cut-off condition is not satisfied.

4.4.3 Approximate KDE for NCA

We recall that NCA was formulated as a class-conditional kernel density estimation problem, Section 3.2. By combining ideas from the previous two subsections, we can develop an NCA specific approximation algorithm.

There are some differences from the classical KDE approximation. In this case, we deal with class-conditional probabilities $p(\mathbf{Ax}_i|c), \forall c$. So each class c needs to be treated separately: we build a k-d tree with the projected data points $\{\mathbf{Ax}_j\}_{j\in c}$ and calculate the estimated probability $\hat{p}(\mathbf{Ax}_i|c)$ for each class. Another distictintion is that for NCA our final interest are the objective function and its gradient. We can easily obtain an approximated version of the objective function by replacing $p(\mathbf{Ax}_i|c)$ with the approximated $\hat{p}(\mathbf{Ax}_i|c)$ in Equation 3.13. To obtain the gradient of this new objective function we can use Equation 3.14. The derivative of $\frac{\partial}{\partial\mathbf{A}}\hat{p}(\mathbf{Ax}|c)$ will be different only for those groups where we do approximations. So, for such a group we obtain:

$$\begin{aligned}\frac{\partial}{\partial\mathbf{A}}\sum_{j\in G}k(\mathbf{Ax}|\mathbf{Ax}_j) &\approx \frac{\partial}{\partial\mathbf{A}}\frac{1}{2}\left\{\min_{j\in G}k(\mathbf{Ax}|\mathbf{Ax}_j)+\max_{j\in G}k(\mathbf{Ax}|\mathbf{Ax}_j)\right\}\\ &=\frac{1}{2}\left\{\frac{\partial}{\partial\mathbf{A}}k(\mathbf{Ax}|\mathbf{Ax}_c)+\frac{\partial}{\partial\mathbf{A}}k(\mathbf{Ax}|\mathbf{Ax}_f)\right\}, \qquad (4.10)\end{aligned}$$

where $\mathbf{Ax}_c$ denotes the closest point in G to the query point $\mathbf{Ax}$ and $\mathbf{Ax}_f$ is the farthest point in G to $\mathbf{Ax}$. Here we made use of the fact the kernel function is a monotonic function of the distance. This means that the closest point gives the maximum contribution, while the farthest point the minimum.

4.5 Exact computations

Exact methods are the counterpart of approximate methods. We can have both efficient and exact computations just by modifying the NCA model. Again, the idea is motivated by the rapid decay of the exponential function. Instead of operating on very small values, we will make them exactly zero. This is achieved by replacing the squared exponential kernel with a compact support function. So, the points that lie outside the support of the kernel are ignored and just a fraction of the total number of points is used for computing the contributions p_{ij}. Further gains in speed are obtained if the search for those points is done with k-d trees (the range search algorithm is suitable for this task; Moore, 1991).

Algorithm 4.6 Approximate NCA objective function and gradient computation

```
Require: Projection matrix $\mathbf{A}$, data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$ and error $\epsilon$.
for all classes $c$ do
    Build $k$-d tree for the points in class $c$.
end for
for all data points $\mathbf{x}_i$ do
    for all classes $c$ do
        Compute estimated probability $\hat{p}(\mathbf{A}\mathbf{x}_i|c)$ and the corresponding derivatives $\frac{\partial}{\partial \mathbf{A}}\hat{p}(\mathbf{A}\mathbf{x}_i|c)$ using approximated KDE Algorithm:
    end for
    Compute soft probability $\hat{p}_i \equiv \hat{p}(c|\mathbf{A}\mathbf{x}_i) = \frac{\hat{p}(\mathbf{A}\mathbf{x}_i|c_i)}{\sum_c \hat{p}(\mathbf{A}\mathbf{x}_i|c)}$.
    Compute gradient $\frac{\partial}{\partial \mathbf{A}}\hat{p}_i$ using Equation 3.14.
    Update function value and gradient value.
end for
```

The choice of the compact support kernel is restricted by a single requirement: differentiability. We will use the simplest polynomial function that has this property. This is given by the following expression:

$$k_{\mathrm{CS}}(u) = \begin{cases} c\ (a^2 - u^2)^2 & \text{if } u \in [-a; +a] \\ 0 & \text{otherwise,} \end{cases} \tag{4.11}$$

where c is a constant that controls the height of the kernel and a is a constant that controls the width of the kernel. In the given context, the kernel will be a function of the distance between two points: $k_{\mathrm{CS}}(u) = k_{\mathrm{CS}}(d_{ij})$, where $d_{ij} = d(\mathbf{A}\mathbf{x}_i; \mathbf{A}\mathbf{x}_j)$. Note that the constant a can be absorbed by the linear projection $\mathbf{A}$. This means that the scale of the learnt metric will compensate for the kernel's width. Also the value for c is not important: from Equation 4.13 we see that this reduces. For convenience, we set both $a = 1$ and $c = 1$. So, we obtain the following simplified version of the kernel:

$$k_{\mathrm{CS}}(d_{ij}) = (1 - d_{ij}^2)^2\ \mathrm{I}(|d_{ij}| \leq 1), \tag{4.12}$$

where $\mathrm{I}(\cdot)$ denotes the indicator function: $\mathrm{I}(\cdot)$ return 1 when its argument is true and 0 when its argument is false.

Now we reiterate the steps of the NCA algorithm (presented in Section 3.1), and replace $\exp(\cdot)$ with $k_{\mathrm{CS}}(\cdot)$. We obtain the following new stochastic neighbour assignments:

$$q_{ij} = \frac{k_{\text{CS}}(d_{ij})}{\sum_{k \neq i} k_{\text{CS}}(d_{ik})}. \tag{4.13}$$

These can be compared to the classical soft assignments given by Equation 3.1. Next we do not need to change the general form of the objective function:

$$f_{\text{CS}}(\mathbf{A}) = \sum_i \sum_{j \in c_i} q_{ij}. \tag{4.14}$$

In order to derive the gradient of the function f_{CS}, we start by computing the gradient of the kernel:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{A}} k_{\text{CS}}(d_{ij}) &= \frac{\partial}{\partial \mathbf{A}} \left[(1 - d_{ij}^2)^2 \cdot \mathrm{I}(|d_{ij}| \leq 1) \right] \\ &= -4\mathbf{A}(1 - d_{ij}^2)\mathbf{x}_{ij}\mathbf{x}_{ij}^{\mathrm{T}} \cdot \mathrm{I}(|d_{ij}| \leq 1), \end{aligned} \tag{4.15}$$

where $\mathbf{x}_{ij} = \mathbf{x}_i - \mathbf{x}_j$.

The gradient of the new objective function is:

$$\frac{\partial f_{\text{CS}}}{\partial \mathbf{A}} = 4\mathbf{A} \sum_{i=1}^{N} \left(q_i \sum_{k=1}^{N} \frac{q_{ik}}{1 - d_{ik}^2} \mathbf{x}_{ik}\mathbf{x}_{ik}^{\mathrm{T}} - \sum_{j \in c_i} \frac{q_{ij}}{1 - d_{ij}^2} \mathbf{x}_{ij}\mathbf{x}_{ij}^{\mathrm{T}} \right). \tag{4.16}$$

This method can be applied in same way as classic NCA: learn a metric $\mathbf{A}$ that maximizes the objective function $f_{\text{CS}}(\mathbf{A})$. Since the function is differentiable, any gradient based method is suitable for optimization and can be used on Equation 4.16.

There is one concern with the compact support version of NCA. There are situations when a point $\mathbf{x}_i$ is placed outside the support of any other point in the data set. Intuitively, this means that the point $\mathbf{x}_i$ is not selected by any point, hence it is not assigned any class label. Also this causes mathematical problems: as in Subsection 3.3.3, the contributions p_{ij} will have an indeterminate value $\frac{0}{0}$. Except of the log-sum-exp trick, the advice from Subsection 3.3.3 can be applied here as well. A more robust way of dealing with this is discussed in the next Section.

4.6 NCA with compact support kernels and background distribution

We extend the previous model to handle cases where points fall outside the support of any other neighbours. The idea is to use for each class a background

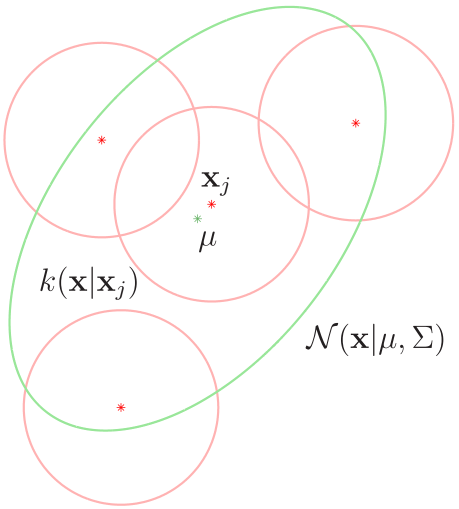

$\mathbf{x}_j$
μ
$k(\mathbf{x}|\mathbf{x}_j)$
$\mathcal{N}(\mathbf{x}|\mu,\Sigma)$

Figure 4.4: Neighbourhood component analysis with compact support kernels and background distribution. The main assumption is that each class is a mixture of compact support distributions $k(\mathbf{x}|\mathbf{x}_j)$ plus a normal background distribution $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu},\boldsymbol{\Sigma})$.

distribution that explains the unallocated points. The background distribution should have an infinite support and an obvious example is the normal distribution.

To introduce a background distribution in a principled manner, we return to the class conditional kernel density estimation (CC-KDE) formulation of NCA, Section 3.2. First, we recast the compact support NCA in the probabilistic framework and consider each class as mixture of compact support distributions:

$$p(\mathbf{x}_i|c) = \frac{1}{N}\sum_{j\in c} k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j), \tag{4.17}$$

where $k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j) = k_{\mathrm{CS}}(d_{ij})$ and is defined by Equation 4.11. Because $k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j)$ denotes a distribution, it ought to integrate to 1. For $c = \frac{15}{16}$ and $a = 1$ the requirement is satisfied.

We further change the model and incorporate an additional distribution in the class-conditional probability $p(\mathbf{x}_i|c)$. From a generative perspective this can be interpreted as follows: a point $\mathbf{x}_i$ is generated by either the compact support distribution from each point $k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j)$ or by a class-specific normal distribution $\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c,\boldsymbol{\Sigma}_c)$. So, the distribution $p(\mathbf{x}_i|c)$ can be written as the sum of these components:

$$p(\mathbf{x}_i|c) = \beta\mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c,\boldsymbol{\Sigma}_c) + (1-\beta)\frac{1}{N_c}\sum_{j\in c} k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j), \tag{4.18}$$

where β is the mixing coefficient between the background distribution and the compact support model, $\boldsymbol{\mu}_c$ is the sample mean of the class c and $\boldsymbol{\Sigma}_c$ is the sample covariance of the class c. The constant β can be set to $\frac{1}{N_c+1}$. This will give equal weights to the background distribution and to each compact support distribution. It might be better to treat β as a parameter and fit it during training. We expect β to adapt to the data set: for example, β should increase for data sets with convex classes.

To finalize this method, we just need to plug Equation 4.18 into the set of Equations 3.11, 3.13 and 3.14. The only difficulty is the gradient computation. We give here only derivatives for each individual component (the full derivations and equations can be found in the Appendix):

- The gradient of the compact support distribution $k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j)$ with respect to $\mathbf{A}$ is very similar to what is given in Equation 4.15. The only difference is that in this case we have everything multiplied by the constant $c = \frac{15}{16}$.

- For the gradient of the background distribution it is useful to note that projecting the points $\{\mathbf{x}_i\}_{i=1}^N$ into a new space $\{\mathbf{A}\mathbf{x}_i\}_{i=1}^N$ will change the sample mean $\boldsymbol{\mu}_c$ to $\mathbf{A}\boldsymbol{\mu}_c$ and the sample covariance $\boldsymbol{\Sigma}_c$ to $\mathbf{A}\boldsymbol{\Sigma}_c\mathbf{A}^{\mathrm{T}}$. Hence, we have:

$$\begin{aligned}\frac{\partial}{\partial \mathbf{A}}\mathcal{N}(\mathbf{A}\mathbf{x}_i|\mathbf{A}\boldsymbol{\mu}_c, \mathbf{A}\boldsymbol{\Sigma}_c\mathbf{A}^{\mathrm{T}}) &= \mathcal{N}(\mathbf{A}\mathbf{x}_i|\mathbf{A}\boldsymbol{\mu}_c, \mathbf{A}\boldsymbol{\Sigma}_c\mathbf{A}^{\mathrm{T}}) \\ &\times \{-(\mathbf{A}\boldsymbol{\Sigma}_c\mathbf{A}^{\mathrm{T}})^{-1}\mathbf{A}\boldsymbol{\Sigma}_c + \mathbf{v}\mathbf{v}^{\mathrm{T}}\mathbf{A}\boldsymbol{\Sigma}_c - \mathbf{v}(\mathbf{x}-\boldsymbol{\mu}_c)^{\mathrm{T}}\}, \end{aligned} \tag{4.19}$$

  where $\mathbf{v} = (\mathbf{A}\boldsymbol{\Sigma}_c\mathbf{A}^{\mathrm{T}})^{-1}\mathbf{A}(\mathbf{x}-\boldsymbol{\mu}_c)$.

- If we also consider β a parameter, we also need the derivative of the objective function with respect to β. This can be easily obtained, if we use the derivative of the class conditional distribution with respect to β:

$$\frac{\partial}{\partial \beta}p(\mathbf{x}_i|c) = \mathcal{N}(\mathbf{x}_i|\boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) - \frac{1}{N_c}\sum_{j\in c} k_{\mathrm{CS}}(\mathbf{x}_i|\mathbf{x}_j). \tag{4.20}$$