Chapter 1

Introduction

k nearest neighbours (kNN) is one of the oldest and simplest classification methods. It has its origins in an unpublished technical report by Fix and Hodges (1951) and since then it has become standard textbook material (Russell et al., 1996; Mitchell, 1997; Bishop, 2006). In the last 50 years, kNN was present in most of the machine learning related fields (pattern recognition, statistical classification, data mining, information retrieval, data compression) and it plays a role in many applications (e.g., face recognition, plagiarism detection, vector quantization).

The idea behind kNN is intuitive and straightforward: classify a given point according to a majority vote of its neighbours; the selected class is the one that is the most represented amongst the k nearest neighbours. This is easy to implement and it usually represents a good way to approach new problems or data sets. Despite its simplicity kNN is still a powerful tool, performing surprisingly well in practice (Holte, 1993).

There are also other characteristics that make kNN an interesting method. First, kNN makes no assumptions about the underlying structure of the data. No a priori knowledge is needed beforehand, but we let the data "speak for itself". The accuracy increases with the number of points in the data set and, in fact, it approaches Bayes optimality as the cardinality of the training set goes to infinity and k is sufficiently large (Cover and Hart, 1967). Second, kNN is able to represent complex functions with non-linear decision boundaries by using only simple local approximations. Lastly, kNN operates in a "lazy" fashion. The training data set is just stored and its use is delayed until testing. The quasi-inexistent training allows the easy addition of new training examples.

kNN has some drawbacks that influence both the computational performance and the quality of its predictions. Since kNN has to store all the exemplars, the memory requirements are directly proportional with the number of instances in the training set. The cardinality of the data also influences the method's speed. All computations are done at testing time, making kNN painfully slow when applied on large data sets.

The accuracy of kNN is closely related to how we define what "close" and "far" mean for our data set and task. Mathematically, the notion of dissimilarity is incorporated into kNN by using different distance metrics. Usually the standard Euclidean metric is not satisfactory and the aim is to find the particular metric that gives the best results on our data set for a given task (section 2.1). There is an entire literature that tries to come up with possible solutions (section 2.2).

1.1 Neighbourhood component analysis

This thesis focuses on neighbourhood component analysis (NCA; Goldberger et al., 2004). The NCA method learns the metric that maximizes the expected number of correctly classified points (section 3.1). Using the NCA metric with kNN usually improves the performance over simple kNN, since we use the label information to construct a suitable metric that selects the relevant attributes. If we restrict the metric to be low ranked we can find its associated linear projection. A low dimensional representation of the original data reduces the storage needs and the computational expense at test time. Also it alleviates some of the concerns that have been raised regarding the usefulness of nearest neighbours methods for high dimensional data (Beyer et al., 1999; Hinneburg et al., 2000). The curse of dimensionality arises for kNN, because the distances become indiscernible for many dimensions. For a given distribution the maximum and the minimum distance between points become equal in the limit of many dimensions. NCA proves to be an elegant answer for the above issues and, consequently, it was successfully used in a variety of applications: face recognition (Butman and Goldberger, 2008), hyper-spectral image classification (Weizman and Goldberger, 2007), acoustic modelling (Singh-Miller, 2010) and even reinformcent learning (Keller et al., 2006).

However, the method introduces additional training time. The objective function needs the pairwise distances of the points, so the function evaluation is quadratic in the number of data points. Also the optimization process is iterative. For large data sets, NCA training becomes prohibitively slow. For example, Weinberger et al. (2006) reported that the original implementation ran out of RAM and Weizman and Goldberger (2007) had to use only 10% of their data set in order to successfully train the model. 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.

1.2 Reducing the computational cost

The main aim of this project is to reduce NCA training time without significant loses in accuracy (chapter 4). We start our investigation with the most simple ideas: use only a subset of the data set (section 4.1) or train the metric on different mini-batches subsampled from the data set (section 4.2). This last idea can be further refined by using clustered mini-batches. Also we present an alternative mini-batch algorithm (section 4.3) that decreases the theoretical cost.

These methods can achieve further speedings if we use approximations of the objective function. Simple approximation ideas (such as ignoring the points which are farther away than a certain distance from the current point) were mentioned in the original paper (Goldberger et al., 2004) and they are re-iterated by Weinberger and Tesauro (2007) and Singh-Miller (2010). We present a more principled approach that borrows ideas from fast kernel density estimation problems (Deng and Moore, 1995; Gray and Moore, 2003; Alexander Gray, 2003). We first recast NCA into a class-conditional kernel density estimation framework (section 3.2) and next we present how fast density estimation is done using a space partitioning structure, such as k-d trees (section 4.4). Similar work of fast metric learning was done by (Weinberger and Saul, 2008) which uses ball trees (Omohundro, 1989) to accelerate a different distance metric technique, large margin nearest neighbour (LMNN; Weinberger et al., 2006). However, LMNN is different from NCA and the way in which ball trees are applied also differs from our approach.

An alternative for the approximation method is to change the model such that it allows exact and efficient computations. In the kernel density estimation model, we can use compact support kernels instead of the standard Gaussian functions. The expense is reduced because only the points that lie in the compact support are used for estimating the density (section 4.5).

We evaluate the proposed techniques in terms of accuracy and speed (chapter 5). Also we provide low dimensional representations of the projected data. The results are promising, showing that we can obtain considerable speed-ups for large data sets while retaining almost the same accuracy as in the classic case. For small and medium sized data sets the accuracy and the time spent are similar to the standard NCA.

Chapter 2

Background

This chapter sets the theoretical basis for the following discussion on low-rank metrics. We start with a brief mathematical introduction into distance metrics. Then we outline the characteristics of the Mahalanobis-like metric. At the end of the chapter we review some of the most prominent work in the area. The method we study, neighbourhood component analysis (NCA), is presented in chapter 3.

2.1 Theoretical background

A distance metric is a mapping from a pair of inputs to a scalar. Let $d(x, y)$ denote the distance function between the arguments x and y. Given a set $\mathcal{S}$, we say that d is a metric on $\mathcal{S}$ if it satisfies the following properties for any $x, y, z \in \mathcal{S}$:

- Non-negativity: $d(x, y) \geq 0$.
- Distinguishability: $d(x, y) = 0 \Leftrightarrow x \equiv y$.
- Symmetry: $d(x, y) = d(y, x)$.
- Triangle Inequality: $d(x, y) \leq d(x, z) + d(z, y)$.

Essentially, a metric can be defined on any set $\mathcal{S}$ of objects. The distance metric is valid as long as the above properties hold for any elements in the set. However, for the scope of our applications we limit ourselves to points in D dimensional real space. The inputs will be represented by column vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$. In this case the distance d is a function from $\mathbb{R}^D \times \mathbb{R}^D$ to $\mathbb{R}$. A well known example of such a function is the Euclidean metric. This is defined by:

$$d(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y})^{\mathrm{T}}(\mathbf{x} - \mathbf{y})}, \quad \text{with } \mathbf{x}, \mathbf{y} \in \mathbb{R}^D. \tag{2.1}$$

Albeit useful in many scenarios, the Euclidean metric has two properties that are problematic in machine learning applications:[1]

- **Sensitivity to variable scaling**. In geometrical situations, the values across all D dimensions are measured in the same unit of length. In practical situations however, each dimension encodes a different feature of the data set. The Euclidean metric is sensitive to scaling and it returns different results if we change the measurement units for one of the D variables. We compensate this differences by dividing the values on dimension i by the standard deviation s_i on that direction:

$$\begin{aligned} d(\mathbf{x},\mathbf{y}) &= \sqrt{\left(\frac{x_1-y_1}{s_1}\right)^2+\cdots+\left(\frac{x_D-y_D}{s_D}\right)^2} = \\ &= \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}\mathbf{S}^{-1}(\mathbf{x}-\mathbf{y})}, \end{aligned} \tag{2.2}$$

  where $\mathbf{S} = \operatorname{diag}(s_1^2, \cdots, s_D^2)$ and $s_i^2 = \operatorname{var}[x_i], \forall i = 1, \cdots, D$.

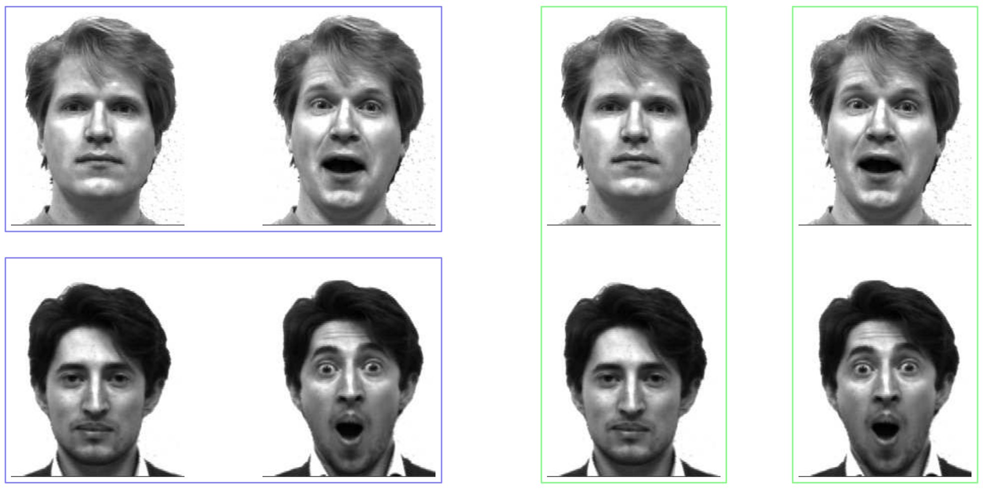
- **Invariance to correlated variables**. Euclidean distance treats equally the discrepancy across any of the D dimensions. Often many features of a data set are redundant and, because of their number, they significantly influence the distance. Take the example of images and face recognition. The pixels from the image background are highly correlated and they reflect the same information: the colour of the background. If the two pictures of the same subject have different backgrounds, we get a high distance between the images. This effect is alleviated if we replace $\mathbf{S}$ in equation (2.2) with the covariance matrix of the data:

$$d(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}\mathbf{S}^{-1}(\mathbf{x}-\mathbf{y})}, \tag{2.3}$$

  where $\mathbf{S} = \operatorname{cov}[\mathcal{D}]$, $\mathcal{D}$ is the dataset. The metric defined in equation (2.3) is known as the Mahalanobis distance.

The Mahalanobis metric extends the Euclidean distance by incorporating data specific information. But a good metric should be even more flexible than that. Apart for the data set, we are also given a task to perform. We want our metric to extract and discriminate only between information that is relevant to our given task. Again, in the face recognition example we are not interested at all in the

[1]`http://matlabdatamining.blogspot.com/2006/11/mahalanobis-distance.html`



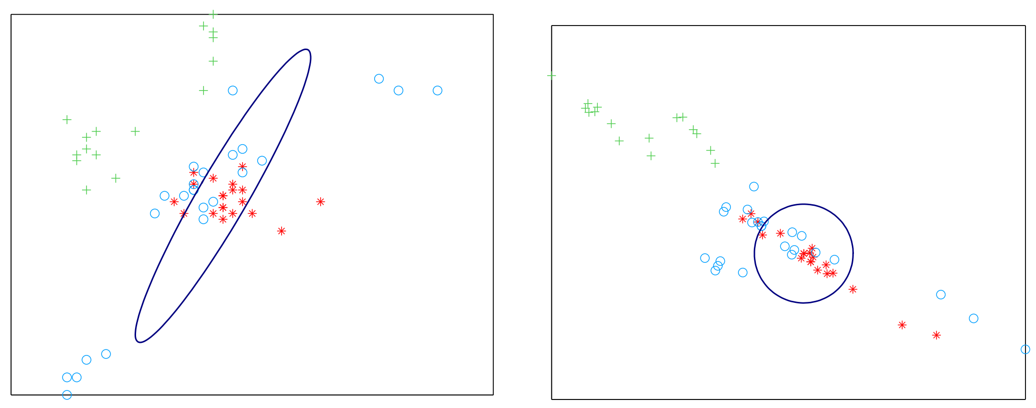
(a) Face recognition (b) Expression recogntion

Figure 2.1: On a given data set we might need to perform multiple tasks; in this example: face recognition and expression recognition. Euclidean metric is not optimal since we want different pairs of equivalence for the two tasks. A good distance metric should be task dependent. These images are an extract from Yale face database.

background colour so the importance of that feature should be down-weighted. In figure 2.1, we illustrate that for a given data set it is not optimal to use a fixed metric.

We can further generalize the result from equation (2.3). Instead of setting $\mathbf{S}$ as the covariance of the data, we let $\mathbf{S}$ be a different matrix. The problem of determining a suitable matrix $\mathbf{S}^{-1}$ is called Mahalanobis (or quadratic) distance metric learning. We note that $\mathbf{S}^{-1}$ ought to be positive semi-definite $\mathbf{x}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{x} \geq 0, \forall\, \mathbf{x}$. This ensures that the norm is a real value: $d(\mathbf{x}, \mathbf{0}) \equiv \|\mathbf{x}\| = \sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{S}^{-1}\mathbf{x}} \in \mathbb{R}$.

There is an interesting equivalence between a Mahalanobis-like metric and a linear transformation. Using Cholesky decomposition we can write $\mathbf{S}^{-1} = \mathbf{A}^{\mathrm{T}}\mathbf{A}$, which gives $\|\mathbf{x}\| = \sqrt{(\mathbf{A}\mathbf{x})^{\mathrm{T}}(\mathbf{A}\mathbf{x})}$, where $\mathbf{A}$ represents a linear transformation of the data. Hence, the problem of finding a suitable distance metric $\mathbf{S}$ is equivalent to the problem of finding a good linear transformation $\mathbf{A}$ and then applying Euclidean distance in the projected space (figure 2.2). We will discuss these two variants interchangeably and we will often parametrize our models in terms of $\mathbf{A}$.



(a) Mahalanobis metric in the original space (b) Euclidean metric in the projected space

Figure 2.2: Illustration of the equivalence between a Mahalanobis metric in the original data space and an Euclidean metric in the transformed data space. The metrics are denoted by an ellipse and, respectively, a circle whose contours indicate equal distance from the centre point to the rest of the points.

2.2 Related methods

The first attempts in metric learning were simple alterations of the Euclidean metric to improve kNN performance. For example, Mitchell (1997) suggests to weight each attribute of the data points such that the cross validation error is minimized. This technique is equivalent to stretching the axis of the relevant attributes and compressing the axis of the attributes that are less informative. The approach is similar to learning a diagonal metric, equation (2.2). Another method (Moore and Lee, 1994) is to eliminate the least relevant attributes: they are given a weight of zero. Albeit useful, these techniques are restrictive as the feature scaling does not reflect how the data covary.

Hastie and Tibshirani (1996) proposed discriminant adaptive nearest neighbours (DANN), a method that constructs a local metric for each particular query. The metric is chosen such that it provides the best discrimination amongst classes in the neighbourhood of the query point. This increases the already costly testing time of kNN. However, DANN method provides a non-linear metric if we take into account all the possible resulted local linear distance metrics. Even though non-linear transformations represent a very interesting domain, we will concentrate our attention on linear metrics as they are less prone to over-fitting and easier to learn and represent.

Xing et al. (2003) proposed learning a Mahalanobis-like distance metric for clustering in a semi-supervised context. There are specified pairs of similar data points and the algorithm finds the linear projection that minimizes the distance between these pairs without collapsing the whole data set to a single point. This approach is formulated as a convex optimization problem with constraints. The solution is attained using an iterative procedure, such as Newton-Raphson. This method assumes that the data set is unimodal and leverages the assumption-free kNN. Another drawback is the training time, because the iterative process is costly for large data sets.

Relevant component analysis (RCA; Bar-Hillel et al., 2003; Shental et al., 2002) is another method that makes use of the labels of the classes in a semi-supervised way to provide a distance metric. More precisely, RCA uses the so-called chunklet information. A chunklet contains points from the same class, but the class label is not known; data points that belong to the same chunklet have the same label, while data points from different chunklets do not necessarily belong to different classes. The main idea behind RCA is to find a linear transformation which "whitens" the data with respect to the averaged within-chunklet covariance matrix (Weinberger et al., 2006). Compared to the method of Xing et al. (2003), RCA has the advantage of presenting a closed form solution, but it has even stricter assumptions about the data: it considers that the data points in each class are normally distributed so they can be described using only second order statistics (Goldberger et al., 2004).

Large margin nearest neighbour (LMNN; Weinberger et al., 2006) is a metric learning method that was designed especially for kNN classification. LMNN can be regarded as the support vector machine (SVM) counter part of metric learning techniques: it brings together points from the same class trying to separate the classes by a certain margin. Also LMNN inherits ths convexity property and falls into the category of semi-definite programming. Weinberger and Saul (2008) worked on fast metric learning by introducing ball trees to speed up the computations. In (Weinberger and Saul, 2008), they extended LMNN to learn local metrics further improving its performance.

Metric learning is an ongoing research area in machine learning with an abundance of different techniques. For a good report that covers many related methods, the reader is referred to (Yang and Jin, 2006).

Chapter 3

Neighbourhood component analysis

This chapter can be regarded as a self-contained tutorial on neighbourhood component analysis (NCA; Goldberger et al., 2004). Much of the material is especially useful for the reader interested in applying the algorithm in practice. We start with a review of the NCA method (section 3.1). Next we recast the NCA model into a class conditional kernel density estimation problem (section 3.2). This new formulation will prove useful later when we want to alter the model in a principled way (section 4.6). Lastly, in section 3.3 we present the lessons learnt from our experience with NCA. Both sections 3.2 and 3.3 offer new insights into the NCA method and they also bring together related ideas from previous work.

3.1 General presentation

Neighbourhood component analysis learns a Mahalanobis metric that improves the performance of k nearest neighbours (kNN). From a practical point of view, NCA is regarded as a desirable additional step before doing classification with kNN. It improves the classification accuracy and it can provide a low-dimensional representation of the data.

Because the goal is to enhance kNN's performance, the first idea Goldberger et al. (2004) had was to maximize the leave one out cross validation performance with respect to a linear projection $\mathbf{A}$. The procedure can be described as follows: apply the linear transformation $\mathbf{A}$ to the whole data set, then take each point $\mathbf{A}\mathbf{x}_i$ and classify it using kNN on the remainder of the transformed data set $\{\mathbf{A}\mathbf{x}_j\}_{j\neq i}$. The matrix $\mathbf{A}$ that achieves the highest number of correct classifications will be used for testing. However, finding the optimal $\mathbf{A}$ is not easy. Any objective function based on the k nearest neighbours is piecewise constant and discontinuous and, hence, hard to optimize. The reason is that there does not exist a direct relation between $\mathbf{A}$ and the neighbours of a given point: a small perturbation in $\mathbf{A}$ might cause strong changes or, conversely, it might leave the neighbours unchanged.

The authors' solution lies in the concept of *stochastic* nearest neighbours. Remember that in the classical 1NN scenario, a query point gets the label of the closest point. In the stochastic nearest neighbour case, the query point inherits the label of a neighbour with a probability that decreases with distance. The stochastic function is reminiscent of the generalized logistic function or of the softmax activation used for neural networks. Let p_{ij} denote the probability that the point j is selected as the nearest neighbour of the point i. The value of p_{ij} is given by:

$$p_{ij} = \frac{\exp(-d_{ij}^2)}{\sum_{k \neq i} \exp(-d_{ik}^2)}, \tag{3.1}$$

where d_{ij} represents the distance between point i and point j in the projected space, i.e. $d_{ij} = d(\mathbf{A}\mathbf{x}_i; \mathbf{A}\mathbf{x}_j) = (\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}(\mathbf{x}_i - \mathbf{x}_j)$. Also we have to set $p_{ii} = 0$: point i cannot pick itself as the nearest neighbour since we assume its label is not known.

Now we can construct a continuous objective function using the stochastic assignments p_{ij} which are differentiable with respect to $\mathbf{A}$. A suitable quantity to optimize is the average probability of each point getting correctly classified. Point i is correctly classified when it is selected by a point j that has the same label as i:

$$p_i = \sum_{j \in c_i} p_{ij}. \tag{3.2}$$

The objective function considers each point in the data set and incorporates their probability of belonging to the true class:

$$\begin{aligned} f(\mathbf{A}) &= \sum_{i=1}^{N} p_i \\ &= \sum_{i=1}^{N} \sum_{j \in c_i} \frac{\exp(-d_{ij}^2)}{\sum_{k \neq i} \exp(-d_{ik}^2)}. \end{aligned} \tag{3.3}$$

The score given by the objective function can be interpreted as the expected number of the correctly classified points.

To obtain the best linear projection $\mathbf{A}$, we maximise $f(\mathbf{A})$ using an iterative gradient based solver such as gradient ascent, conjugate gradients or delta-bar-delta (subsection 3.3.1). For these methods, we can use the analytical expression of the gradient. If we differentiate with respect to $\mathbf{A}$, we obtain:

$$\frac{\partial f}{\partial \mathbf{A}} = 2\mathbf{A}\sum_{i=1}^{N}\left(p_i \sum_{k=1}^{N} p_{ik}\mathbf{x}_{ik}\mathbf{x}_{ik}^{\mathrm{T}} - \sum_{j\in c_i} p_{ij}\mathbf{x}_{ij}\mathbf{x}_{ij}^{\mathrm{T}}\right), \tag{3.4}$$

where $\mathbf{x}_{ij} = \mathbf{x}_i - \mathbf{x}_j$. The interested reader can find the derivation in the appendix.

At test time, we use the learnt matrix $\mathbf{A}$ to transform the points before doing classification with kNN. Given a query point $\mathbf{x}^*$, we assign it the label c_j of the closest point j in the projected space, where $j = \operatorname{argmin}_i d(\mathbf{A}\mathbf{x}^*, \mathbf{A}\mathbf{x}_i)$.

Further remarks

There are some interesting observations that can be made about NCA. It is useful to note that NCA model is still valid even if we set the matrix $\mathbf{A}$ to be rectangular, i.e. $\mathbf{A} \in \mathbb{R}^{d\times D}$, with $d < D$. In this case the linear transformation $\mathbf{A}$ is equivalent to a low dimensional projection. Restricting $\mathbf{A}$ to be non-square gives rise to a couple of advantages. Firstly, the optimization process is less prone to overfitting since a rectangular matrix has fewer parameters than a square one. Secondly, the low dimensional representation of the data set reduces the memory requirements and the computational cost at test time. In this thesis we concentrate mostly on low-rank metrics partly motivated by the above reasons, but also because many techniques that are subsequently developed (chapter 4) work best in low dimensions.

We need to underline that NCA's objective function is not convex. The first consequence is that the parameter space is peppered with local optima and care must be taken to avoid poor solutions. Factors such as the choice of initialization or the optimization procedure affect the quality of the final result. Section 3.3 discusses these practical issues in detail. The other side of non-convexity is that it allows NCA to cope well with data sets that have multi-modal class densities. This is an important advantage over classical methods such as principal component analysis, linear discriminant analysis or relevant component analysis. All of these assume that the data is normally distributed which harms the classification performance on complicated data sets.

Another important characteristic of NCA is its computational cost. For each point, we need to calculate the distances to all the other points. So the number of operations is quadratic in the number of data points. The evaluation of the objective function is done in two steps. First we find the point projections $\{\mathbf{A}\mathbf{x}_i\}_{i=1}^N$; this operation has $\mathcal{O}(dDN)$ cost. Next we compute all the distances d_{ij} in the low dimensional space; this is done in $\mathcal{O}(dN^2)$ flops. The total cost is then $\mathcal{O}(dDN + dN^2)$. Another way of evaluating the function f is to parametrize it in terms of the Mahalanobis metric $\mathbf{S} = \mathbf{A}^\mathrm{T}\mathbf{A}$. Then we can compute the distances in the D-dimensional space in $\mathcal{O}(DN^2)$ flops. This option is slower than the previous case for $d < D$ and $D \ll N$.

Since the optimization is based on the gradient, we are also interested in its cost. The evaluation of gradient given in the original paper and in equation (3.4) scales in D^2N^2. We improve this by "pushing" the matrix $\mathbf{A}$ into the sum as in (Singh-Miller, 2010):

$$\frac{\partial f}{\partial \mathbf{A}} = 2\sum_{i=1}^{N}\left(p_i \sum_{k=1}^{N} p_{ik}(\mathbf{A}\mathbf{x}_{ik})\mathbf{x}_{ik}^\mathrm{T} - \sum_{j\in c_i} p_{ij}(\mathbf{A}\mathbf{x}_{ij})\mathbf{x}_{ij}^\mathrm{T}\right), \tag{3.5}$$

If we consider that the projections $\mathbf{A}\mathbf{x}_i$ were already calculated for the function evaluation, the gradient has a cost of $\mathcal{O}(dDN^2)$. However, in theory, evaluating the gradient should have the same complexity as evaluating the objective function. Automatic differentiation (AD; Rall, 1981) is a technique that achieve this theoretical result. AD uses the chain rule to derivative the objective function and split the gradient into elementary functions that can be readily computed. So instead of evaluating the complicated analytical gradient, we work on a series of cheap operations. This method is very useful in practice and there exist libraries for this in most programming languages. In our implementation we used the analytical differentiation, because we were not aware of AD until recently.

The computational drawback represents an important setback in applying NCA in practice. For example, Weinberger and Tesauro (2007) reported that the method ran out of memory on large data sets and Weizman and Goldberger (2007) used only 10% of an hyper-spectral data set to train the NCA classifier. We present a number of solutions to this problem in chapter 4.

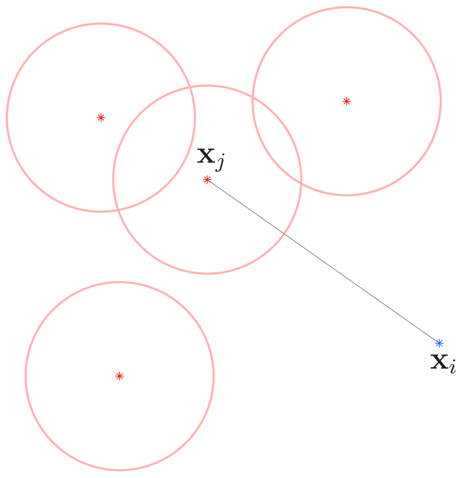


Figure 3.1: NCA as a class-conditional kernel density estimation model. The figure represents a schematic illustration of a mixture of Gaussians. The red circles denote isotropic Gaussians. These are centred onto the points that belong to a certain class. A point i is generated by either of the components of the class with a probability inverse proportional with the distance.

3.2 Class-conditional kernel density estimation interpretation

In this section we recast NCA into a class-conditional kernel density estimation framework. This interpretation will allow us to understand the assumptions behind NCA method. After this we are in position to alter the model in a suitable way that is efficient for computations (sections 4.4 and 4.5). Similar ideas were previously presented by Weinberger and Tesauro (2007) and Singh-Miller (2010), but the following were derived independently and they offer different insights. Our following interpretation was inspired by the probabilistic kNN presented by Barber (2011).

We start with the basic assumption that each class can be modelled by a mixture of Gaussians. For each of the N_c data points in class c we consider a Gaussian "bump" centred around it (figure 3.1). From a generative perspective, we can imagine that each point j can generate a point i with a probability given by an isotropic normal distribution with variance σ^2:

$$\begin{aligned} p(\mathbf{x}_i|\mathbf{x}_j) &= \mathcal{N}(\mathbf{x}_i|\mathbf{x}_j, \sigma^2 \mathrm{I}_D) \qquad (3.6) \\ &= \frac{1}{(2\pi)^{D/2}\sigma^D} \exp\left\{ -\frac{1}{2\sigma^2}(\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}}(\mathbf{x}_i - \mathbf{x}_j)\right\}. \qquad (3.7) \end{aligned}$$

By changing the position of the points through a linear transformation $\mathbf{A}$, the probability changes as follows:

$$p(\mathbf{A}\mathbf{x}_i|\mathbf{A}\mathbf{x}_j) \propto \exp\left\{ -\frac{1}{2\sigma^2}(\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}(\mathbf{x}_i - \mathbf{x}_j)\right\}. \qquad (3.8)$$

We note that this is similar to the p_{ij} from NCA. Both $p(\mathbf{A}\mathbf{x}_i|\mathbf{A}\mathbf{x}_j)$ and p_{ij} are directly proportional with the same quantity.

Using the mixture of Gaussians assumption, we have that the probability of a point of being generated by class c is equal to the sum of all Gaussians in class c:

$$\begin{aligned} p(\mathbf{x}_i|c) &= \frac{1}{N_c}\sum_{\mathbf{x}_j \in c} p(\mathbf{x}_i|\mathbf{x}_j) \qquad (3.9) \\ &= \frac{1}{N_c}\sum_{\mathbf{x}_j \in c} \mathcal{N}(\mathbf{x}_i|\mathbf{x}_j, \mathrm{I}_D). \qquad (3.10) \end{aligned}$$

However, we are interested in the inverse probability, given a point $\mathbf{x}_i$ what is the probability of $\mathbf{x}_i$ belonging to class c. We obtain the expression for $p(c|\mathbf{x}_i)$ using Bayes' theorem:

$$p(c|\mathbf{x}_i) = \frac{p(\mathbf{x}_i|c)p(c)}{p(c|\mathbf{x}_i)} = \frac{p(\mathbf{x}_i|c)p(c)}{\sum_c p(\mathbf{x}_i|c)p(c)}. \qquad (3.11)$$

Now if we further consider the classes to be equal probable (which is a reasonable assumption if we have no a priori information) we arrive at result that resembles the expression of p_i (equation (3.2)):

$$p(c|\mathbf{A}\mathbf{x}_i) = \frac{\frac{1}{N_c}\sum_{\mathbf{x}_j \in c} \exp\left\{-\frac{1}{2\sigma^2}(\mathbf{x}_i - \mathbf{x}_j)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}(\mathbf{x}_i - \mathbf{x}_j)\right\}}{\frac{1}{N_{c'}}\sum_{c'}\sum_{\mathbf{x}_k \in c'} \exp\left\{-\frac{1}{2\sigma^2}(\mathbf{x}_i - \mathbf{x}_k)^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}(\mathbf{x}_i - \mathbf{x}_k)\right\}} \qquad (3.12)$$

We are interested in predicting the correct class of the point i. So we try to find that linear transformation $\mathbf{A}$ that maximises the class conditional probability of this point $p(c_i|\mathbf{A}\mathbf{x}_i)$ to its true class c_i.

$$f(\mathbf{A}) = \sum_i p(c_i|\mathbf{A}\mathbf{x}_i). \qquad (3.13)$$

.

As in section 3.1, we can optimize the function using the gradient information. The gradient is the following:

$$\frac{\partial f}{\partial \mathbf{A}} = \sum_i \left\{ \frac{\frac{\partial p(\mathbf{A}\mathbf{x}_i|c)}{\partial \mathbf{A}} p(c)}{\sum_c p(\mathbf{x}_i|c)p(c)} - \underbrace{\frac{p(\mathbf{A}\mathbf{x}_i|c)p(c)}{\sum_c p(\mathbf{A}\mathbf{x}_i|c)p(c)}}_{p(c|\mathbf{A}\mathbf{x}_i)} \frac{\sum_c \frac{\partial p(\mathbf{A}\mathbf{x}_i|c)}{\partial \mathbf{A}} p(c)}{\sum_c p(\mathbf{A}\mathbf{x}_i|c)p(c)} \right\}. \tag{3.14}$$

The main advantage of the above formulation is that we can construct different models simply by modifying the class-conditional probability (equation (3.9)). Once $p(\mathbf{x}_i|c)$ is altered in a suitable way, we use Bayes rule in conjuction equations (3.13) and (3.14) to get the new function and its corresponding gradient. We optimize for parameter $\mathbf{A}$ exactly in the same way as for classical NCA.

Examples of this technique are in sections 4.5 and 4.6. There the alterations allowed considerable speed-ups as we shall see in chapter 5.

Other benefits of CC-KDE model are the possibility to incorporate prior information about the class distributions $p(c)$ and the possibility to classify a point $\mathbf{x}^*$ using $p(c|\mathbf{A}\mathbf{x}^*)$ (subsection 3.3.5).

3.3 Practical notes

While NCA is not that hard to implement (appendix A.1), certain care is needed in order to achieve good solutions. In this section we try to answer some of the questions that can be raised when implementing NCA.

3.3.1 Optimization methods

The function $f(\mathbf{A})$, equation (3.3), can be maximised using any gradient based optimizer. We considered two popular approaches for our implementation: gradient ascent and conjugate gradients. These are only briefly presented here. The interested reader is pointed to (Bishop, 1995).

Gradient ascent

Gradient ascent is one of the simplest optimisation methods. It is an iterative algorithm that aims to find the maximum of a function by following the gradient direction at each step. The entire procedure is summarized by algorithm 3.1.

Besides this version, an alternative is to compute the gradient on a fewer points and not on the entire data set. This variant is known as stochastic gradient ascent and it has the advantage of being much faster than the batch version. Different alterations of the stochastic gradient ascent were tried in chapter 4 to reduce the computational cost.

Algorithm 3.1 Gradient ascent (batch version)

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$.
1: Get initial $\mathbf{A}_0$
2: repeat
3:     Update parameter: $\mathbf{A}_{t+1} \leftarrow \mathbf{A}_t + \eta \frac{\partial f(\mathbf{A}_t, \mathcal{D})}{\partial \mathbf{A}}$
4:     Update learning rate $\eta$
5:     $t \leftarrow t + 1$
6: until convergence
7: return $\mathbf{A}_t$.
```

For the algorithm 3.1, there are three aspects we need to carefully consider:

Initialization. The algorithm starts the search in the parameter space from the an initial parameter $\mathbf{A}_0$. Different values for $\mathbf{A}_0$ will give different final solutions. In the NCA context, initialization is related to finding a good initial linear transformation; this is discussed separately in subsection 3.3.2. For the moment, we can assume the values of $\mathbf{A}_0$ are randomly chosen.

Learning rate. The parameter η is called the *step size* or, in the neural networks literature, it also known as the *learning rate.* As name suggests, η controls how much we move the parameters in the gradient direction.

The learning rate can be either fixed or adaptive. For the first case, choosing the correct value for η is critical. If η is set too large, the algorithm diverges. On the other hand, a small η results in slow convergence.

Usually, a variable learning rate η is preferred since it is more flexible. The "bold driver" (Vogl et al., 1988) is a reasonable heuristic for automatically modifying η during training. The idea is to gradually increase the learning rate after each iteration as long as the objective function keeps improving. Then we go back to the previous position and start decreasing the learning rate until a new increase in the function is found. The common way to increase the step size η is by multiplying it with a constant $\rho > 1$, usually $\rho = 1.1$. To decrease the step size, we can multiple it with a constant $\sigma < 1$, for example $\sigma = 0.5$.

Another way is to change the learning rate η using a decreasing rule proportional to the number of current iteration: $\eta = \frac{\eta_0}{t+t_0}$. Now we have two parameters instead of one. Intuitively, η_0/t_0 can be regarded as the initial learning rate and t_0 as the number of iterations after which η_0 starts to drop off. Using a learning rate of such form is especially motivated in the stochastic gradient case. If $\eta \propto 1/t$, the algorithm converges to the true minimum as t goes to infinity. However, in practice the convergence is very slow; so setting the constants η_0 and t_0 is important for reaching a good solution in a short time. Leon Bottou[1] recommends setting η_0 to be the regularization value and then select t_0 such that the resulting update will modify $\mathbf{A}$ by a sensible amount. Since we did not use regularization, we followed some of the tricks presented in (LeCun et al., 1998). We fixed the value for η_0 and we did an exponential search for t_0 using a cross validation set.

Convergence. The gradient ascent algorithm can be stopped either when we hit a maximum number of iterations or when the learning rate η falls below a certain threshold. Also we can look at the variation in the objective function. If the value of the objective function stops improving we can halt the algorithm. Another popular choice to check convergence is "early stopping". We monitor the performance of the new $\mathbf{A}$ on a cross validation set. If the objective function did not increase for more than a preset number of iterations we stop and return to the previous best solution. This solution prevents overfitting since we stop when we achieve a minimum on a test error and not on the training error. We illustrate this in figure 3.2.

Gradient ascent has known convergence problems and determining good learning rates is often difficult. But it has the advantage of being quite flexible, especially in the stochastic optimization scenario, where we can select any noisy estimate of the gradient in order to replace the total gradient.

[1]Oral communication: `http://videolectures.net/mmdss07_bottou_lume/`

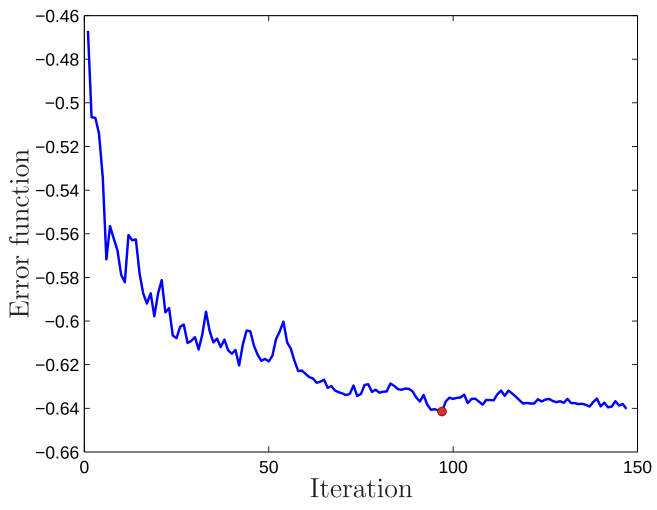


Figure 3.2: This figure illustrates the early stopping procedure. During training we keep a small part of the data set apart, for cross-validation. On this cross-validation data set we compute the error function at each iteration. When the error function stops decreasing we stop the learning procedure and return to the parameter that achieved the best value. This figure resulted while applying NCA on the `mnist` data set. The best value was obtained at iteration 98, as indicated by the red dot.

Conjugate gradients

The conjugate gradient (CG; Hestenes and Stiefel, 1952) method solves some of the convergence issues. Instead of selecting the search directions to be the gradient directions, the CG method selects the next direction depending on both the previous directions and on the curvature of the surface. CG reaches the maximum of a quadratic energy function in a number of iterations equal to the number of dimensions the function has the support on. Also CG eliminates the manual set-up for the learning rates. As Bishop (1995) notes, CG can be regarded as a more complex version of gradient ascent that automatically chooses the parameters. The details of the conjugate gradient method are out of the scope of this thesis; a gentle introduction into the subject is given by Shewchuk (1994).

In practice it is considerably easier to use an existing implementation of CG (for example, Carl E. Rasmussen's `minimize.m`[2]) than trying to find the optimal parameters for the learning rate for gradient ascent. For small and medium-sized data sets, the "bold driver" heuristic gives comparable scores to the more

[2]Source code can be found at the following URL: `http://www.gaussianprocess.org/gpml/code/matlab/util/minimize.m`

sophisticated CG method (tables B.3 and B.4).

3.3.2 Initialization

Initialization is important because the function $f(\mathbf{A})$ is not convex. The quality of the final solution relies on the starting point of the optimisation algorithm. A general rule of thumb is to try multiple initial seeds and select the final $\mathbf{A}$ that gives the highest score.

We have already mentioned random initialization in subsection 3.3.1. Beside this, we can try linear transformations that are cheap to compute. Such examples include principal component analysis (PCA; Pearson, 1901), linear discriminant analysis (LDA; Fisher, 1936) and relevant component analysis (RCA; Bar-Hillel et al., 2003). For completeness, we give here the equations and also include further notes:

- PCA finds an orthogonal linear transformation of the data. This is obtained by computing the eigendecomposition of the outer covariance matrix:

$$\mathbf{S} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^{\mathrm{T}}. \tag{3.15}$$

- LDA finds a linear transformation $\mathbf{A}$ by maximizing the variance between classes $\mathbf{S}_B$ relative to the amount of within-class variance $\mathbf{S}_W$:

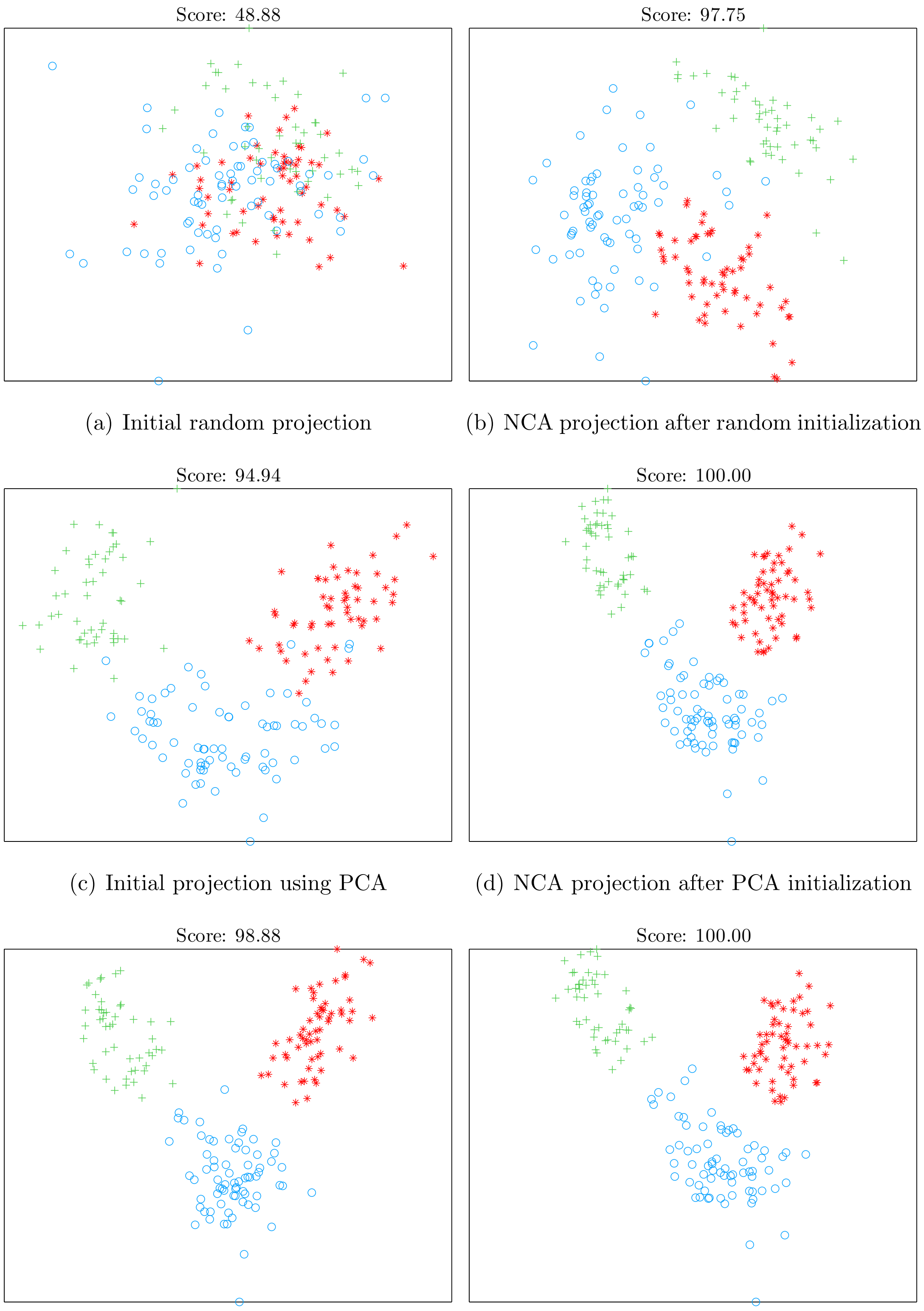
$$\mathbf{S}_B = \frac{1}{C}\sum_{c=1}^{C}\boldsymbol{\mu}_c\boldsymbol{\mu}_c^{\mathrm{T}} \tag{3.16}$$

$$\mathbf{S}_W = \frac{1}{N}\sum_{c=1}^{C}\sum_{i\in c}(\mathbf{x}_i - \boldsymbol{\mu}_c)(\mathbf{x}_i - \boldsymbol{\mu}_c)^{\mathrm{T}}. \tag{3.17}$$

  The projection matrix $\mathbf{A}$ that achieves this maximization consists of the eigenvectors of $\mathbf{S}_W^{-1}\mathbf{S}_B$.

  Unlike PCA, LDA makes use of the class labels and this usually guarantees a better initial projection.

- RCA finds a linear transformation $\mathbf{A}$ that decorrelates the points within a chunklet, i.e. it makes the within-chunklet covariance equal to the identity matrix. Because for NCA we restrict ourselves to fully labelled data, the within-chunklet covariance is the within-class covariance $\mathbf{S}_W$, equation (3.17). The "whitening" transformation $\mathbf{A}$ is then $\mathbf{A} = \mathbf{S}_W^{-1/2}$.



(a) Initial random projection

(b) NCA projection after random initialization

(c) Initial projection using PCA

(d) NCA projection after PCA initialization

(e) Initial projection using LDA

(f) NCA projection after LDA initialization

Figure 3.3: Results for different initializations (random, PCA and LDA) on `wine` data set. The images on the left side represent the projections of the data set using the initial $\mathbf{A}$. The images on the right side are data set projection with the final $\mathbf{A}$. Above each figure there is presented the LOOCV score which is normalized to 100.

If the projection is full-rank, $\mathbf{A} \in \mathbb{R}^{D \times D}$, other obvious initializations are the identity matrix $\mathbf{A} = \mathrm{I}_D$ and the Mahalanobis linear transformation $\mathbf{A} = \mathbf{S}^{-1/2}$.

If we want to learn a low-rank projection, $\mathbf{A} \in \mathbb{R}^{d \times D}, d < D$, then we can still use the eigendecomposition based methods. We construct $\mathbf{A}$ using only the top d most discriminative eigenvectors, *i.e.*, those eigenvectors that have the highest eigenvalues associated.

From our experiments, we conclude that a good initialization results in a good solution and a faster convergence; these benefits are more evident on large data sets. As advertised by Butman and Goldberger (2008), we found RCA to work the best. Figure 3.3 depicts initialization effects on a small data set. More illustrations are in the appendix **??**. Occasionally random initialization gives better final scores than the other techniques (figure B.2).

3.3.3 Numerical issues

Numerical problems can easily appear when computing the soft assignments p_i. If a point $\mathbf{x}_i$ is far away from the rest of the points, the stochastic probabilities $p_{ij}, \forall j$, are all 0 in numerical precision. Consequently, the result p_i is undetermined: $\frac{0}{0}$. To give an idea of how far $\mathbf{x}_i$ has to be for this to happen, let us consider an example in MATLAB. The answer to `exp(-d^2)` is `0` whenever `d` exceeds `30` units. This is problematic in practice, since distances larger than 30 often appear. Some common cases include data sets that contain outliers or data sets that present a large scale.

The large scale effect can be partially alleviated if we initialize $\mathbf{A}$ with small values. This idea was used by Laurens van der Maaten in his implementation: `A = 0.01*randn(d,D)`. However, this does not guarantee to compensate the scale variation for any data set.

A more robust solution is to normalize the data, i.e., centre and make it have unit marginal variances:

$$x_{ij} \leftarrow \frac{x_{ij} - \mu_j}{\sigma_j}, \quad i = \{1, \cdots, N\},\ j = \{1, \cdots, D\}. \tag{3.18}$$

In this case, we have to store the initial mean and variance of the data and, at test time, transform the test data accordingly: subtract the mean and scale it using the variance coefficients. The variance scaling can be regarded as a linear transformation. We can combine this transformation with the learnt transformation

A and get:

$$\mathbf{A}_{\text{total}} \leftarrow \mathbf{A} \cdot \begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_D \end{pmatrix}. \tag{3.19}$$

In general, data normalization avoids numerical issues for the first iterations. But during training, the scale of **A** increases and data points can be "thrown" away. We adopt a rather simplistic approach to avoid any numerical problems: replace p_i with a very small value whenever we are in the $\frac{0}{0}$ case, as in Laurens van der Maaten implementation. In Matlab, this is done using the following command `max(p_i,eps)`.

A more rigorous way of dealing with this by multiplying both the numerator and denominator of p_i with a certain quantity $\exp(L)$:

$$p_i = \frac{\sum_{j \in c_i} \exp(L - d_{ij}^2)}{\sum_{k \neq i} \exp(L - d_{ik}^2)}, \tag{3.20}$$

where $L = \min_{k \neq i} d_{ik}^2$. This value of L ensures that at least one term in the denominator does not underflow.

In our implementation, we preferred the first trick because we can vectorize the code. In `Matlab`, it is important to have vectorized code to achieve better speed-ups. For a C implementation, we recommend the second option.

3.3.4 Regularization

Although the original paper (Goldberger et al., 2004) claimed that there were no problems with overfitting, we observed that NCA objective function has the tendency to increase the scale of the linear projection **A**. Butman and Goldberger (2008) pointed out that this is especially problematic for data sets whose size N is significantly smaller than the dimensionality D. In such a case, the linear transformation **A** can be chosen to project each point $\mathbf{x}_i$ to a pre-specified a location $\mathbf{y}_i$. The values of **A** are obtained by solving the linear system:

$$\mathbf{A}\mathbf{x}_i = \mathbf{y}_i, \quad i = \{1, \cdots, N\}. \tag{3.21}$$

Because the system has more equations dN than unknowns dD, it has exact solutions. If we set the same positions $\mathbf{y}_i$ for points in the same class, we can virtually increase the scale of the projection to infinity and get an error-free classification. This degeneracy can be corrected with the help of regularization. We alter the objective function by subtracting a regularization term proportional to the magnitude of $\mathbf{A}$. This gives:

$$g(\mathbf{A}) = f(\mathbf{A}) - \lambda \sum_{i=1}^{d} \sum_{j=1}^{D} A_{ij}^2. \tag{3.22}$$

$$\frac{\partial g}{\partial \mathbf{A}} = \frac{\partial f}{\partial \mathbf{A}} - 2\lambda \mathbf{A}, \tag{3.23}$$

where λ is a positive constant that is tuned via cross-validation.

Another effect of a large scale linear projection $\mathbf{A}$ is the fact that only the nearest neighbour is considered for each point. This is not usually the optimal thing to do, so regularization is also used to prevent this (Singh-Miller, 2010).

In our implementation, we use "early stopping" technique which has a similar effect to regularization.

3.3.5 Doing classification

For testing, Goldberger et al. (2004) considered only the kNN decision rule. Since we are optimizing a particular function $f(\mathbf{A})$, another sensible tool for classification is an NCA based function. If we are using the probabilistic interpretation (section 3.2) a query point $\mathbf{x}_i$ is labelled with most probable class c:

$$c = \operatorname{argmax}_c p(c|\mathbf{x}_i). \tag{3.24}$$

This can be re-written in NCA specific notation as:

$$c = \operatorname{argmax}_c \frac{\sum_{j \in c} p_{ij}}{\sum_k p_{ik}}. \tag{3.25}$$

In our experiments, 1NN and the NCA classification rule described by equation (3.25) give very similar results if we train $\mathbf{A}$ om the un-regularized function $f(\mathbf{A})$. When we used early stopping, the NCA classification rule yielded better performances (chapter 5). In this case, kNN with $k > 1$ should also perform better than simple 1NN.

3.3.6 Dimensionality annealing

Dimensionality annealing is an approach to learning a low-rank metric in an way that avoids local optima (Hinton via Murray, oral communication). We start with a full rank projection **A** and gradually reduce the rank by introducing regularization terms on the lines of **A**. Compared with the classical regularization procedure, subsection 3.3.4, here we use a regularization term on each dimension d. The new objective function and its gradient are given by the following equations:

$$g(\mathbf{A}) = f(\mathbf{A}) - \sum_{i=1}^{d} \lambda_i \sum_{j=1}^{D} A_{ij}^2 \tag{3.26}$$

$$\frac{\partial g}{\partial \mathbf{A}} = \frac{\partial f}{\partial \mathbf{A}} - 2 \begin{pmatrix} \lambda_1 A_{11} & \cdots & \lambda_1 A_{1D} \\ \vdots & \ddots & \vdots \\ \lambda_d A_{d1} & \cdots & \lambda_d A_{dD} \end{pmatrix}. \tag{3.27}$$

A large value of λ_d will impose a small magnitude of **A** on dimension d. We increase λ_d slowly; this permits the rest of the values of **A** to readjust. We clarify these steps in algorithm 3.2.

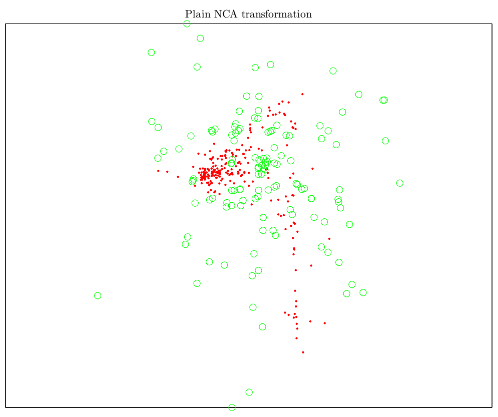
Algorithm 3.2 Dimensionality annealing

```
Require: Data set $\mathcal{D} = \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$, initial linear transformation $\mathbf{A} \in \mathbb{R}^{D \times D}$
    and low dimension $d$.
1: for $i = 1, \cdots, D - d$ do
2:     Select dimension $d$ to anneal
3:     for $j = 1, \ldots, P$ do
4:         Increase regularization coefficient $\lambda_d \leftarrow \lambda_d + \Delta\lambda$
5:         Optimize function $g$: $\mathbf{A} = \mathbf{A} + \eta \frac{\partial g(\mathbf{A}, \mathcal{D}, \boldsymbol{\lambda})}{\partial \mathbf{A}}$
6:     end for
7: end for
```

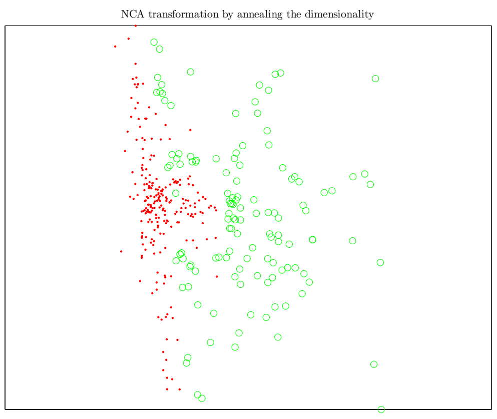
There are several notes to be made. There are alternatives for selecting the dimension d we want to anneal, step 2 of algorithm 3.2. We can choose d to be the direction that has:

- minimum variance in our data set $\{\mathbf{Ax}_i\}_{i=1}^N$
- minimum variance in the projected data set $\{\mathbf{Ax}_i\}_{i=1}^N$
- smallest magnitude in **A**.

The function optimization step can be done either by gradient ascent or conjugate gradients. It is possible to do conjugate gradients for a small number of iterations, typically 2 or 3.



(a) Simple NCA transformation



(b) NCA with dimensionality annealing projection

Figure 3.4: Example of applying the dimensionality annealing procedure to `ionosphere` data set. The method using dimensionality annealing is more visually appealing than the classical approach.

After a dimension is annealed, we reach the end of the inner for loop, we can remove that particular dimension; for example set the elements on the d row equal to 0.

A further idea would be to run conjugate gradients until convergence initialized with low dimensional $\mathbf{A}$ returned by algorithm 3.2.

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.

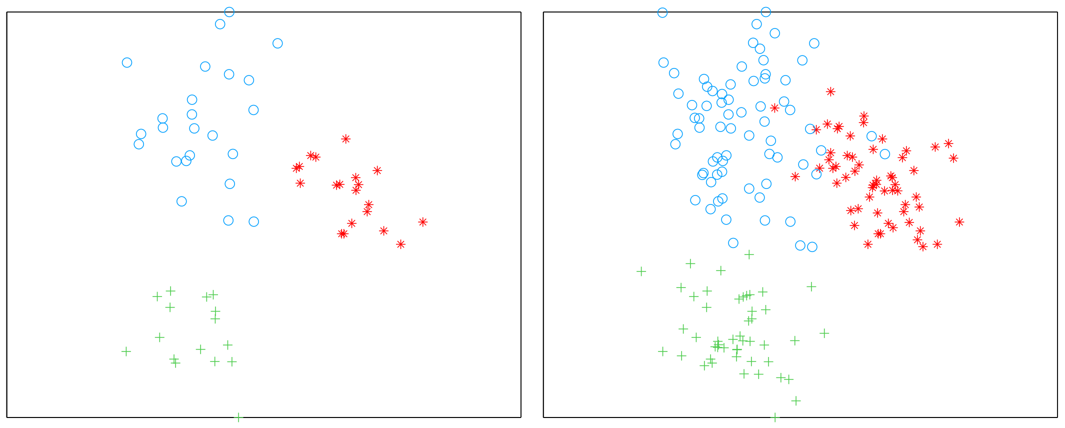
This chapter aims to complete the existing work. We present a series of methods that can improve NCA's speed. Most of these ideas are new in the context of NCA. They are also quite versatile. Each method proposes a new objective function and can be regarded as an independent model on its own.

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.
1: Randomly pick a point that will be the first centre $\mathbf{c}_1$.
2: Allocate all the points in the first cluster $\mathcal{M}_1 \leftarrow \mathcal{D}$.
3: for $i = 1$ to $K$ do
4:     Select the $i$-th cluster centre $\mathbf{c}_i$ as the point that is farthest away from any
       cluster centre $\mathbf{c}_1, \cdots, \mathbf{c}_{i-1}$.
5:     Move to the cluster $\mathcal{M}_i$ those points that are closer to its centre than to
       any other cluster centre: $\mathcal{M}_i = \{\mathbf{x} \in \mathcal{D} \mid d(\mathbf{x}; \mathbf{c}_i) < d(\mathbf{x}; \mathbf{c}_j), \forall j \neq i\}$
6: 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.
1: if $N < n$ then
2:     New cluster: $i \leftarrow i + 1$.
3:     return current points as a cluster: $\mathcal{M}_i \leftarrow \mathcal{D}$.
4: else
5:     Randomly select two points $\mathbf{x}_j$ and $\mathbf{x}_k$ from $\mathcal{D}$.
6:     Project all data points onto the line defined by $\mathbf{x}_j$ and $\mathbf{x}_k$. (Give equation?)
7:     Select the median value $\tilde{\mathbf{x}}$ from the projected points.
8:     Recurs on the data points above and below $\tilde{\mathbf{x}}$: RPC($\mathcal{D}_{>\tilde{\mathbf{x}}}$) and RPC($\mathcal{D}_{\leq\tilde{\mathbf{x}}}$).
9: 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:

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

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

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

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.
1: repeat
2:   Split data $\mathcal{D}$ into groups $\mathcal{M}_i$ of size $n$.
3:   for all $\mathcal{M}_i$ do
4:     Update parameter using gradient given by Equation 4.2:
5:     $\mathbf{A} \leftarrow \mathbf{A} + \eta \frac{\partial f_{\text{sNCA}}(\mathbf{A}, \mathcal{M}_i)}{\partial \mathbf{A}}$.
6:     Update learning rate $\eta$.
7:   end for
8: 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 $\frac{\partial p_{N+1}}{\partial \mathbf{A}}$.

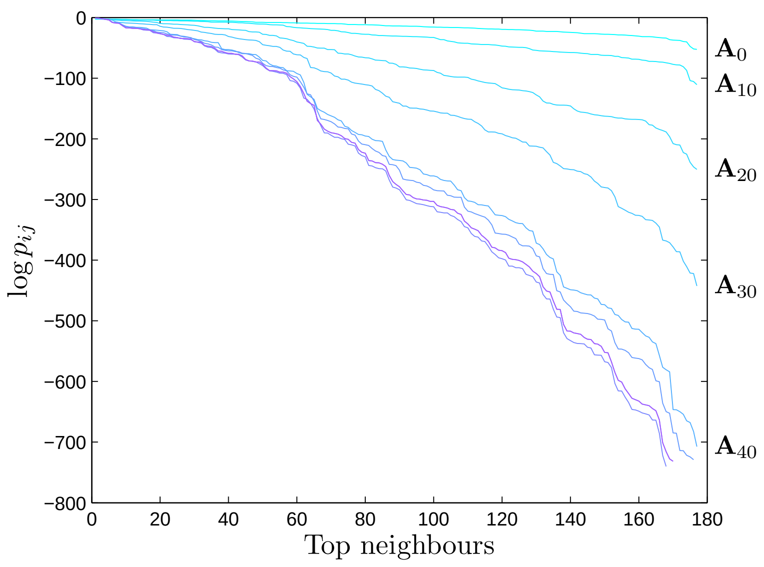


Figure 4.2: Evolution of the stochastic assignments p_{ij} during training for a given point $\mathbf{x}_i$.

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 $\mathbf{x}_i$.

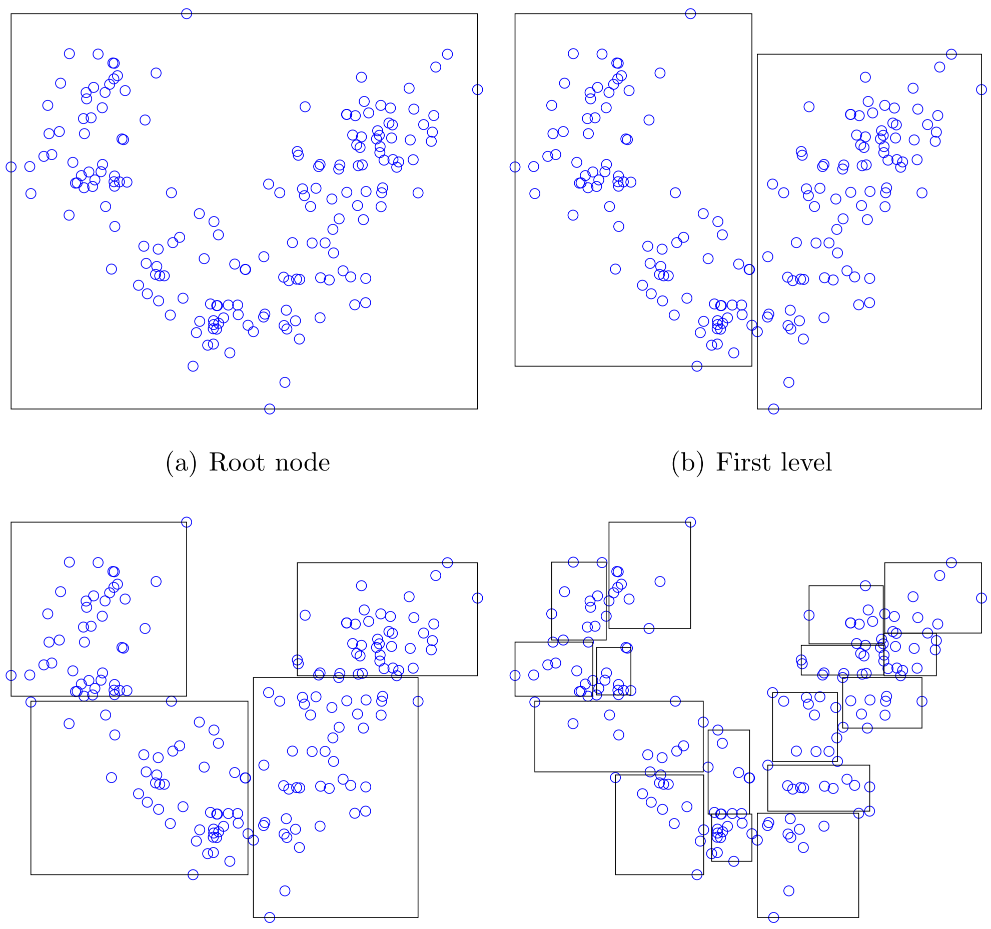
4.4.1 k-d trees

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:

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

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.
 1: if $N < m$ then
 2:     Mark node $i$ as leaf: splitting_direction(i)=-1.
 3:     Add points to leaf: points(i)$\leftarrow \mathcal{D}$.
 4:     return
 5: end if
 6: Choose direction $\vec{d}$ using Equation 4.4: splitting_direction(i)=$d$.
 7: Find the median value $\tilde{x}_d$ on the direction $\vec{d}$: splitting_value(i)=$\tilde{x}_d$.
 8: 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.
 9: Build left child build_kdtree($\mathcal{D}_{\leq \tilde{\mathbf{x}}}$,2*i).
10: 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 $p_{\text{SoFar}}(\mathbf{x}_i) + \max k(\mathbf{x}_i|\mathbf{x}_j)$. Hence, a possible cut-off rule is:

Note that in this case it is important the order in which we accumulate. A large $p_{\text{SoFar}}(\mathbf{x}_i)$ 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 $\epsilon$.
 1: for all classes $c$ do
 2:     Build $k$-d tree for the points in class $c$.
 3: end for
 4: for all data points $\mathbf{x}_i$ do
 5:     for all classes $c$ do
 6:         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:
 7:     end for
 8:     Compute soft probability $\hat{p}_i \equiv \hat{p}(c|\mathbf{A}\mathbf{x}_i) = \frac{\hat{p}(\mathbf{A}\mathbf{x}_i|c_i)}{\sum_c \hat{p}(\mathbf{A}\mathbf{x}_i|c)}$.
 9:     Compute gradient $\frac{\partial}{\partial \mathbf{A}}\hat{p}_i$ using Equation 3.14.
10:     Update function value and gradient value.
11: 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:

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

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: $k_{\text{CS}}(u) = k_{\text{CS}}(d_{ij})$, 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:

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

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 $k_{\text{CS}}(\cdot)$. 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:

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

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

The gradient of the new objective function is:

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

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

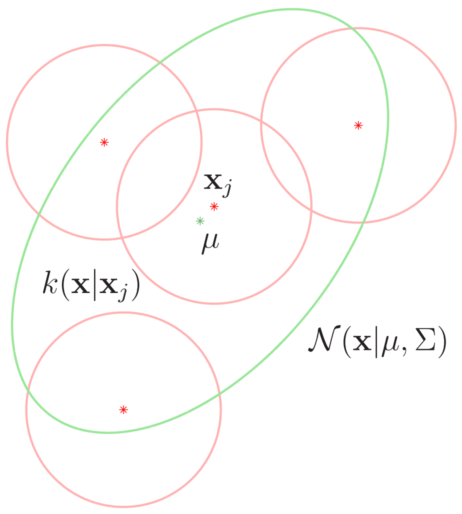

$\mathbf{x}_j$
μ
$k(\mathbf{x}|\mathbf{x}_j)$
$\mathcal{N}(\mathbf{x}|\mu,\Sigma)$

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:

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

Chapter 5

Evaluation

This chapter contains the evaluation of the methods proposed in the previous part, chapter 4. In section 5.1, we present the data sets and the methodology used for testing. We give details about the parameter choices and set baseline scores obtains by either classical NCA or simple linear projections, such as PCA, LDA or RCA. Results for each individual method are presented in sections 5.3 and 5.4. A comparison of the methods is shown in subsection 5.3.4 using accuracy versus time plots.

We should mention that we did not include all the results in this chapter to prevent cluttering. Further experimentations can be found in appendix B.

5.1 Evaluation setup

The evaluation was done in terms of two metrics: accuracy and speed. An additional and more subjective criterion is to judge a method by visualizing low dimensional representations of various data sets. We provided 2D plots of the projected data where suitable.

The data sets selected for testing are listed in table 5.1. We note that the used data vary both as number of samples N and as dimensionality D. Even if we concentrate on large amounts of data, we need small data sets to assess the performance of the new models. The methods' speed was tested on the large data sets (`usps`, `magic` and `mnist`). However, the diversity in size and complexity made it difficult to find the optimal selection of parameters. We are aware that there is "no free lunch" in accurately solving widely different problems with a fixed model. When concentrating on a single task it is often advised to include

Data set name	Abbrevation	N	D	C
Balance scale	`balance`	625	4	3
Ecoli	`ecoli`	336	7	8
Glass identification	`glass`	214	9	6
Ionosphere	`ionosphere`	351	33	2
Iris	`iris`	150	4	3
Landsat satellite	`landsat`	6435	36	6
MAGIC Gamma telescope	`magic`	19020	10	2
MNIST digits	`mnist`	70000	784	10
Pima Indians diabetes	`pima`	768	8	2
Image segmentation	`segment`	2310	18	7
SPECTF heart	`spectf`	267	44	2
Blood transfusion	`transfusion`	748	4	2
USPS digits	`usps`	11000	256	10
Wine	`wine`	178	13	3
Yeast	`yeast`	1484	8	10

Table 5.1: This table presents the characteristics of the data sets used: number of samples N, dimensionality of the data D and number of classes C. The two digits data sets `mnist` and `usps` were downloaded from the following URL `http://cs.nyu.edu/~roweis/data.html`. All the others data sets are available in the UCI repository `http://archive.ics.uci.edu/ml/datasets.html`.

prior knowledge into the model. Also it is easier to make minor tweaks of the parameters to boost the performance.

For evaluation we used 70% of the data set for training and the rest of 30% was kept for testing. We made exceptions for two data sets: `landsat` and `mnist`. These are commonly already split in training and testing sets. We report standard errors for the mean accuracy averaged over different splits.

The methods we test are: sub-sampling (SS; section 4.1), mini-batches (MB; section 4.2), stochastic learning (SL; section 4.3). For stochastic learning we included the two approximated methods: the fast kernel density estimation idea (SL-KDE; section 4.4) and compact support version of NCA (SL-CS; section 4.5). Each method has particular parameters that we discuss in its corresponding subsection. There are some common parameters for all the methods. These choices are related to the NCA implementation and, for convenience, we remind them here. We experimented with three optimization methods: gradient ascent with "bold driver" heuristic, conjugate gradients and variants of stochastic gradient ascent with early stopping. For initialization we used the techniques described in subsection 3.3.2: random initialization, PCA, LDA or RCA. At test time, we did classification using 1-NN or using an NCA based function as described in subsection 3.3.5. However, we usually present scores using both classification rules.

The experiments were carried in MATLAB and most of the implementations are the authors' own work. There are some exceptions however. We used Carl E. Rasmussen's `minimize.m` for conjugate gradient optimization.[1] The RCA implementation was provided by Noam Shental on his web-page.[2] Also we used functions from Iain Murray's MATLAB toolbox.[3] Finally, we inspected previous implementations of NCA,[4] even if we did not explicitly made use of them.

5.2 Baseline

We started by implementing the standard NCA algorithm (appendix A.1). This consists the main baseline against which we compare new models. For our first series of experiments, we tried to replicate the work in the original article (Goldberger et al., 2004). We encountered some difficulties since no information about their implementation was provided in the paper. Our results are presented in table 5.2 (scores are averaged over 40 runs). We randomly initialized the matrix $\mathbf{A}$ and optimized it using conjugate gradients (CG) method. This was the easiest thing to do since no parameter tuning is need for CG. We note that the results are similar to those of Goldberger et al. (2004); for `ionosphere` data set we obtained slightly worse results.

However, in order to achieve a robust implementation of NCA we had to carry out additional experiments. The learnt lessons were summarized in section 3.3.

[1]This is available for download at <`http://www.gaussianprocess.org/gpml/code/matlab/util/minimize.m`>.

[2]The code was downloaded from <`http://www.openu.ac.il/home/shental/`>.

[3]Iain Murray's toolbox is available at <`http://homepages.inf.ed.ac.uk/imurray2/code/imurray-matlab/`>.

[4]Implementations of NCA are provided by Laurens van der Maaten in his MATLAB Toolbox for Dimensionality Reduction <`http://homepage.tudelft.nl/19j49/Matlab_Toolbox_for_Dimensionality_Reduction.html`> and by Charless C. Fowlkes on his website <`http://www.ics.uci.edu/~fowlkes/software/nca/`>.

		Train score	Test scores		Baseline
Data set	d	$f(\mathbf{A})$	1-NN	NCA	Eucl.
`balance`	2	92.86 ± 0.47	90.78 ± 0.53	90.61 ± 0.55	
	$D = 4$	95.36 ± 0.38	93.40 ± 0.47	93.04 ± 0.51	76.18
`ionosphere`	2	98.31 ± 0.14	79.86 ± 0.75	79.74 ± 0.78	
	$D = 33$	72.07 ± 0.71	86.22 ± 0.64	72.87 ± 0.71	85.38
`iris`	2	99.38 ± 0.11	94.94 ± 0.39	94.72 ± 0.39	
	$D = 4$	99.48 ± 0.10	95.10 ± 0.44	95.15 ± 0.44	95.53
`wine`	2	99.15 ± 0.14	92.4 ± 1.0	92.4 ± 1.0	
	$D = 13$	98.95 ± 0.15	95.36 ± 0.51	95.36 ± 0.51	74.53

Table 5.2: Accuracy of standard NCA on four small data sets. Scores are averaged over 40 runs. The second column presents the dimensionality d the data set is reduced to. The last column shows the leave one out cross validation performance on the data set using Euclidean metric.

Data set	PCA	LDA	RCA
`usps`	73.47 ± 0.13	87.44 ± 0.12	87.42 ± 0.13
`magic`	77.097 ± 0.080	76.17 ± 0.29	77.574 ± 0.078
`mnist`	70.13	9.96	79.11

Table 5.3: Accuracy of three linear transformation techniques applied on the large data sets. We used 1-NN for classification. Scores are averaged over 20 runs, except for `mnist` data set. We reduced each data set dimensionality to $d = 5$.

We will further see the influence of the implementation tricks in the next part (section 5.3). Results were shown in section 3.3 and are also attached at the end of the thesis, appendix B:

- Tables B.3 and B.4 compare two of the optimization methods: conjugate gradients and gradient ascent with "bold driver" heuristic. We observe that the two methods give close scores on most of data sets. However, the gradient ascent takes longer until it reaches convergence.
- Figures B.1, B.2 and B.3 illustrate the initialization effect on three data sets: `iris`, `balance` and `ecoli`. RCA seems to be the best option for initialization and we used it in most of our comparisons. However, random projection can also be sometimes surprisingly good as we see in figure B.1(a).

We could not apply NCA on large data sets (`usps`, `magic`, and `mnist`) since it would have taken far too long. We decided to use as a baseline the linear transformations that we also use for initialization (PCA, LDA and RCA). The results are averaged over different splits of training and testing set and they are listed in table 5.3. For `mnist` data set we have scores of classic NCA (Singh-Miller, 2010). For $d = 5$ the accuracy obtained was 91.1% using kNN classifier and 90.9% using NCA classifier. For the classification procedure Singh-Miller learnt the optimal value for k using cross-validation.

5.3 Mini-batches methods

The mini-batches methods are compared on the larger data sets: `usps`, `magic`, and `mnist`. We chosen to reduce the dimensionality to $d = 5$. The decision is partially motivated by the fact NCA is very effective for reducing the data dimensionality to small values, somewhere around 5 to 15. A more principled approach would have been to develop a model choosing algorithm: start with a projection to $d = 2$ dimensions and then increase the number of dimensions until the score stops improving. The idea is similar to the "early stopping" procedure and it is illustrated for `landsat` data set in figure 5.1.

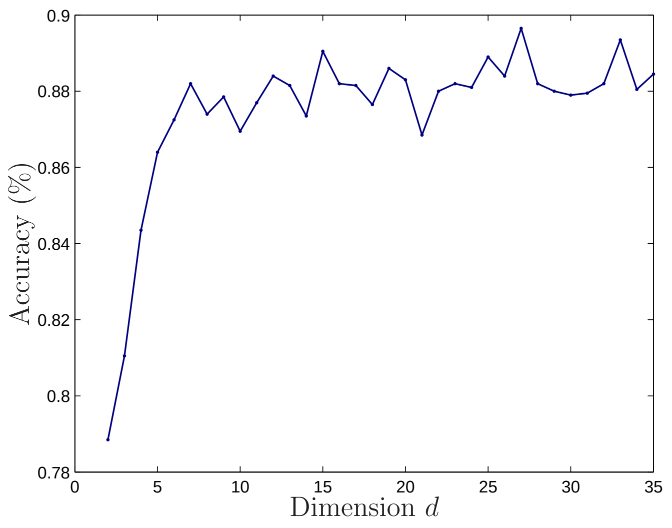


Figure 5.1: Evolution of test accuracy as dimensionality increases on `landsat` data set. We see that NCA operates almost as well in low dimensions ($d = 6, \cdots, 10$) as in high dimensions ($d > 25$). This approach can be used for selecting a suitable dimension to which we project the data.

Data set	Train	1-NN	NCA
`usps`	99.112 ± 0.099	89.30 ± 0.18	89.41 ± 0.16
`magic`	82.71 ± 0.41	79.25 ± 0.45	80.22 ± 0.72
`mnist`	96.867	82.130	82.470

Table 5.4: Accuracy scores for SS method on the larger data sets. We used RCA for initialization and CGs for optimization. We used a subset of $n = 3000$ data points for training and the whole data set for testing.

5.3.1 Sub-sampling

For sub-sampling we trained NCA on a subset of $n = 3000$ samples of the original data. We used conjugate gradients for optimization and RCA linear transformation for initialization. As previously mentioned in section 4.1, the sub-sampled data has a thinner distribution than the original data which helps the method to obtain good scores at training. But the test performance is hindered because we do not use the true distribution. This is especially evident for the digits data `usps` and `mnist` (table 5.3.1).

Data set	Train	1-NN	NCA
`usps`	92.00 ± 0.43	91.26 ± 0.17	92.37 ± 0.17
`magic`	80.04 ± 0.48	79.14 ± 0.52	79.8 ± 1.1
`mnist`	87.47 ± 0.49	87.52 ± 0.36	89.88 ± 0.25

Table 5.5: Accuracy scores for MB method on the larger data sets. We used RCA for initialization and the mini-batches were clustered in the low-dimensional space using RPC. The size of a mini-batch was of maximum $n = 2000$ data points.

5.3.2 Mini-batches

We trained NCA using the gradient ascent variant with clustered mini-batches (section 4.2). For the learning rate, we used an update rule of the form $\frac{\eta}{t+t_0}$. We fixed $\eta = 1$ and tuned the other free parameter t_0 using cross validation across an exponential scale from 0.1 to 1000. After that, a finer tuning was done on a linear scale around the best value of t_0. We used 5% of the training set for cross validation to monitor the accuracy score at each iteration. If the performance on the cross validation set does not increase for 25 iterations, we stop the learning process and return to the previously best parameter.

To get significant gradients we used large mini-batches $n = 2000$. The points in a batch are selected via recursive projection clustering (RPC) algorithm. The clustering was done in the low dimensional space, after projecting the points with the current linear transformation matrix $\mathbf{A}$. The results obtained by mini-batches method can be found in table 5.3.2. We note similar scores on `magic` and better results on the other two data sets.

5.3.3 Stochastic learning

This method was trained using the variant of stochastic gradient ascent presented in section 4.3. We used the same parameters as in the previous section. We considered $n = 50$ neighbours to look at for each iteration and computed their contributions with respect to the whole data set.

Besides the results on the large data sets (table 5.3.3), we also present the performance of this method when used for small data sets (table 5.3.3). We note a considerable improvement on the `magic` data set compared to the previous two methods. For small data sets, we observe similar results as the baseline NCA.

		Train score	Test scores	
Data set	d	$f(A)$	1-NN	NCA
`balance`	2	88.35 ± 0.83	87.37 ± 0.49	90.45 ± 0.38
	$D = 4$	94.70 ± 0.87	95.32 ± 0.34	96.14 ± 0.29
`ionosphere`	2	89.0 ± 1.7	85.71 ± 0.94	87.08 ± 0.95
	$D = 33$	92.6 ± 1.5	84.72 ± 0.57	84.34 ± 0.59
`iris`	2	96.41 ± 0.94	96.33 ± 0.57	97.00 ± 0.46
	$D = 4$	97.5 ± 1.3	95.67 ± 0.71	96.11 ± 0.66
`wine`	2	98.80 ± 0.70	97.22 ± 0.65	97.50 ± 0.49
	$D = 13$	99.25 ± 0.62	96.85 ± 0.41	96.85 ± 0.41

Table 5.6: Accuracy scores for SL method on the small data sets. We used RCA for initialization. The scores are averaged after 20 iterations.

Data set	Train	1-NN	NCA
`usps`	90.23 ± 0.50	90.68 ± 0.22	92.64 ± 0.17
`magic`	78.39 ± 0.25	79.76 ± 0.13	84.49 ± 0.12
`mnist`	85.97 ± 0.37	86.07 ± 0.43	89.35 ± 0.39

Table 5.7: Accuracy scores for SS method on the larger data sets. We used RCA for initialization. At each iteration we consider $n = 50$ data points.

The most important remark is that the classification done using NCA objective function is better than 1-NN classification. This observation also applies for the large data sets results. More results for this method are in appendix, tables B.1 and B.2.

5.3.4 Comparison

To easily compare the 3 presented methods, we provide time-accuracy plots (figure 5.2). We did not compute time for smaller data sets, since the classical NCA was usually fast enough. We also included in the plots the method based on the compact support kernels (we discuss its individual scores in subsection 5.4.2).

In terms of accuracy, all of the NCA variants improve the accuracy over PCA, LDA or RCA. SS gives comparable results to the other two only on `magic` data set. This data set has only two classes and a sub-sampled data set does not remove too much information of the decision boundaries. MB and SL are similar in accuracy. We note the proposed methods obtain a slightly worse score than classic NCA on `mnist`. The performance reported in (Singh-Miller, 2010) was around 91%, while the mean accuracy for MB and SL is about 90%.

The differences in time varied strongly. We used a logarithmic scale on the time to better discriminate the plotted scores. For each method we show the equidensity contours of the Gaussian distribution. The contours are ellipses, but because of the logarithmic axis they look deformed.

The time spent varies even for the same method because the number of iterations until convergence depends on random parameters. There is a certain trade-off between time and accuracy, especially for the digits data sets.

We show also low dimensional representations of data for $d = 2$ for two of the data sets: `usps` (figure 5.3) and `magic` (figure 5.4).

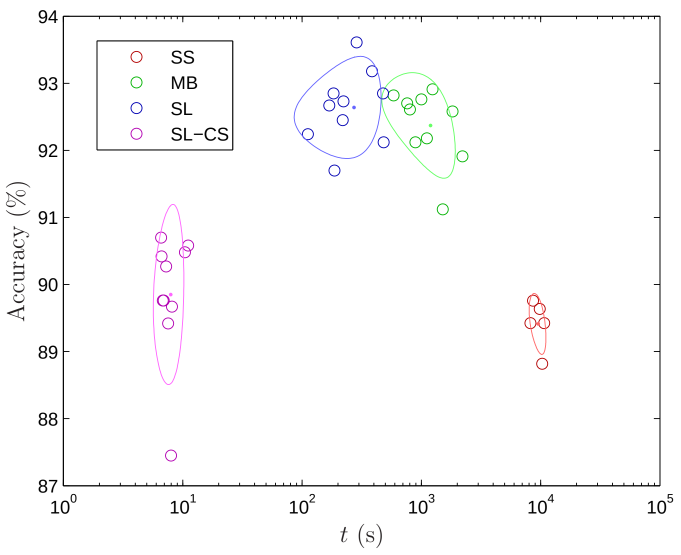
5.4 Approximate computations

The following methods can be applied on classic NCA, but we tested them in conjunction with the stochastic learning procedure to further boost the speed.

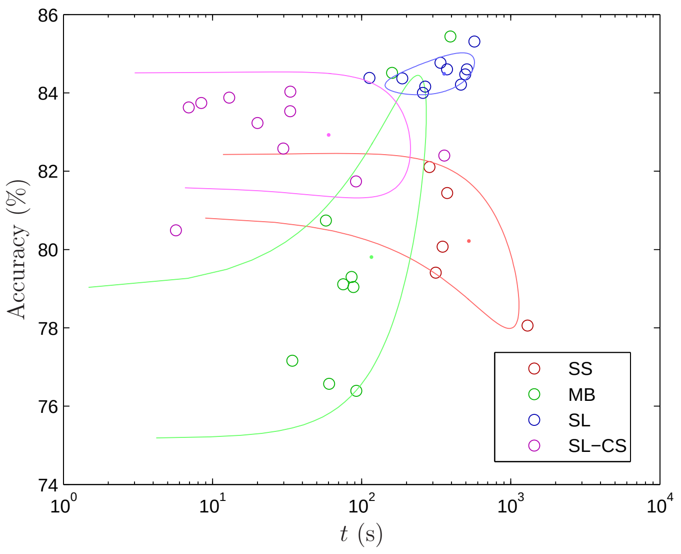
For approximate computations we also tried a more simplistic approach, similar to that of Weinberger and Tesauro (2007): we selected only the first 100 neighbours for each point and discarded those points whose contribution p_{ij} is less than $\exp(-30)$. This simple approach gave surprinsingly good score, although we do not present them here.

5.4.1 NCA with k-d trees

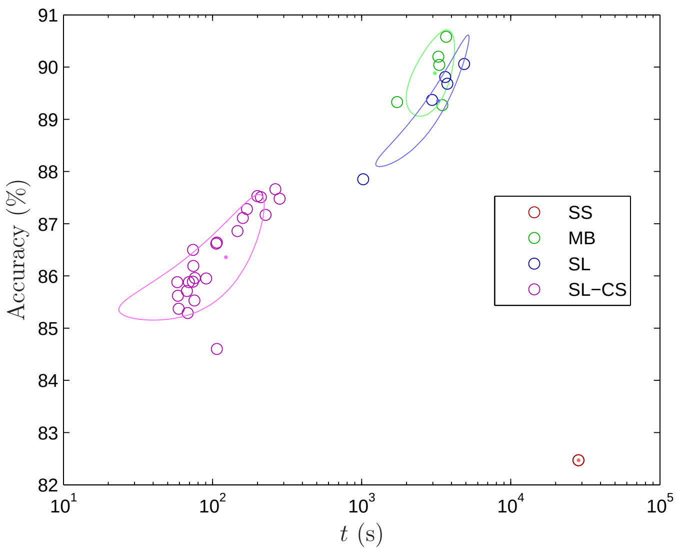
For the class-conditional kernel density estimation idea we used the algorithm described in section 4.4. Our k-d tree implementation was done in MATLAB and, for this reason, it was pretty slow. The code was slower than the simple SL version, because the we cannot vectorize the recursive computation of the objective function and its gradient. We experimented with kernel density code written in C and mex-ed in MATLAB and we think that such an approach can improve the speed. Doing simple kernel density estimation with a C written code



(a) usps

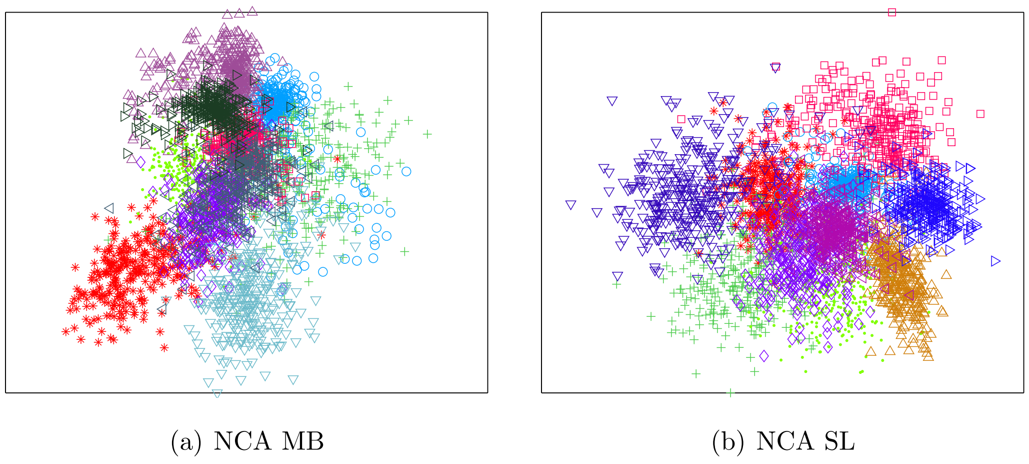


(b) magic



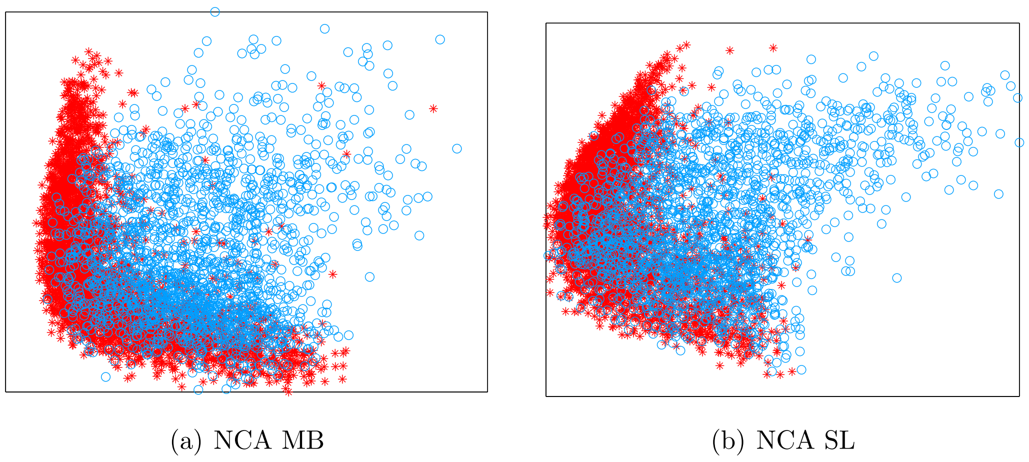
(c) mnist

Figure 5.2: Time *vs.* accuracy plots on larger data sets for four of the proposed methods. For SS we plotted the 1-NN score, while for the other three the points inidicate the NCA score.



(a) NCA MB (b) NCA SL

Figure 5.3: Two dimensional projections of `usps` data set using two variants of NCA learning. The linear transformation was learnt on a training set, and here is plotted the projection of a testing set.



(a) NCA MB (b) NCA SL

Figure 5.4: Two dimensional projections of `magic` data set using two variants of NCA learning. The linear transformation was learnt on a training set, and here is plotted the projection of a testing set.

$\epsilon_{\max}$	Train	1-NN	NCA	Visited points
0	87.05 ± 0.66	86.30 ± 0.32	87.87 ± 0.24	80.0 ± 5.9
10^{-50}	87.50 ± 0.31	86.26 ± 0.24	87.88 ± 0.20	55.1 ± 5.2
10^{-20}	85.49 ± 0.89	86.32 ± 0.28	87.87 ± 0.26	45.7 ± 4.1
10^{-5}	87.29 ± 0.49	85.95 ± 0.19	87.63 ± 0.29	22.8 ± 2.7
0.01	87.34 ± 0.76	86.14 ± 0.29	87.90 ± 0.24	22.1 ± 4.2
0.1	86.70 ± 0.39	86.12 ± 0.29	88.09 ± 0.19	20.0 ± 2.0

Table 5.8: NCA SL + k-d trees on `landsat`. $\epsilon_{\max}$ denotes the maximum error that we accept while approximating the density for a point given a class $p(\mathbf{x}|c)$. Visited points indicates the fraction of points that are used for computing the function and the gradient.

Data set	Train	1-NN	NCA
`usps`	89.68 ± 0.51	90.57 ± 0.20	92.67 ± 0.15
`magic`	78.09 ± 0.37	79.68 ± 0.30	84.50 ± 0.21
`mnist`	84.4 ± 1.4	85.5 ± 1.0	88.98 ± 0.75

Table 5.9: NCA SL + k-d trees on large data sets. For these experiments we set $\epsilon_{\max} = 0.1$.

proved to be even faster than doing it in MATLAB. The short time did not permit us to finalize this approach. We demonstrate that SL with k-d trees achieves good results even if we discard a large number of the points. Table 5.9 shows results for this method on the large data sets. We also present how the accuracy and the average number of prunings depend on the maximum error imposed (table 5.8).

5.4.2 NCA with compact support kernels

The compact support version of NCA is easy to implement and it is very fast as we already saw in section 5.3.4. Usually only a small fraction of the points is inspected. In figure 5.5, we see how this fraction evolves with the learning process. In the first iterations there might be a larger number of points, but soon this drops off and stabilizes. It might be tempting to start with all the points very close together to ensure that no point is outside the compact support of any

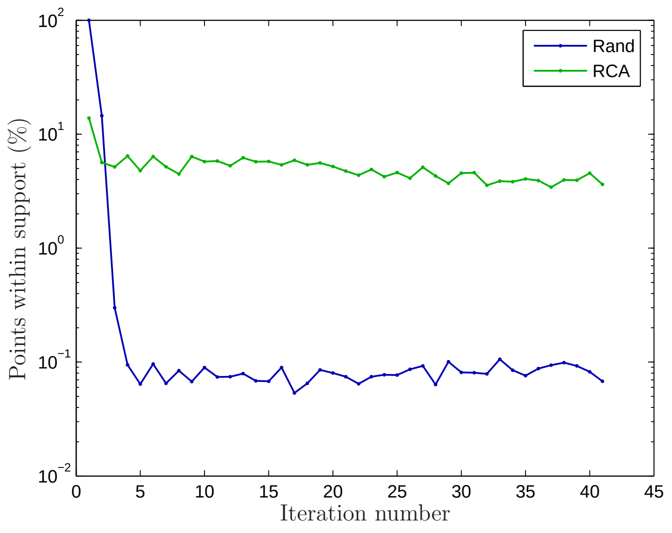
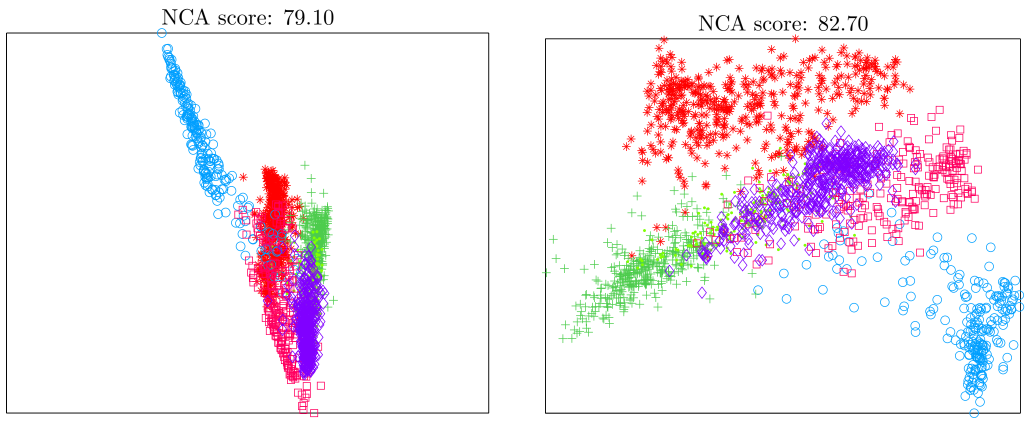


Figure 5.5: This figure illustrates how the fraction of the points inspected varies during the learning procedure. When we use random initialization, there are inspected only 0.1% of the points. If RCA is used to initialize the learning algorithm a fraction of about 10% is used.



(a) NCA SL CS randomly initialized (b) NCA SL CS with RCA initialization

Figure 5.6: Illustration of final projections using two different initializations for NCA SL CS.

Data set	Train	1-NN	NCA
`usps`	85.16 ± 0.33	87.02 ± 0.44	89.85 ± 0.30
`magic`	76.92 ± 0.56	79.09 ± 0.42	82.92 ± 0.36
`mnist`	80.55 ± 0.28	81.96 ± 0.26	86.36 ± 0.18

Table 5.10: Accuracy scores for NCA SL CS method on the larger data sets.

other point. This approach has however two drawbacks. Since all the points are in the support of all the other points it means the first iteration will not present any gain in speed. In fact, the first iteration can be easily as expensive as the whole learning process. A second issue is that the gradients will be very large and might "throw away" points out of the existing support. More reliable is an initialization with the linear transformation such as RCA. This provides a more stable evolution. We also see that the final projection looks better in the second case (figure 5.6).

Results that demonstrate performance in terms of accuracy on small data sets are attached in appendix, tables B.5 and B.6. The comparison has as baseline the classical NCA and the extended version of NCA with compact support kernels and background distribution (NCA CS BACK). The results on the large data set are available in table 5.10.

5.5 Recommendations

When dealing with a large data set, we suggest to first try NCA SL CS. This method has the advantage of being easy to implement and very fast even for large data sets. However, the speed does come at a cost. We note a slight decrease in accuracy for most experimentations. Nonetheless, applying NCA SL CS first gives us an idea of how well NCA can perform on a given data set. If one is pleased with the score, it can use the classic NCA SL or the more sophisticated NCA SL + k-d tree and hope to get an improvement of a couple of percentages in accuracy. However all the methods offer better scores than the eigendecomposition based methods.

Chapter 6

Conclusions

This thesis presented our approach to fast low-rank metric learning. The need for a low rank metric was motivated in the context of kNN (chapter 1 and section 2.1), but we argue that learning a metric is useful whenever our algorithm relies on dissimilarities. Our efforts were directed towards the already established method, neighbourhood component analysis (NCA; section 3.1). We introduced a family of algorithms to mitigate NCA's main drawback — the computational cost. In our attempt of speeding up NCA, we encountered other interesting theoretical and practical challenges. The answers to these issues represent an important part of the thesis (sections 3.2 and 3.3). We summarize here our main contributions and present the conclusions.

Section 3.2 The class-conditional kernel density estimation (CC-KDE) interpretation offers flexibility to NCA and opens a new range of methods. We can easily change the model in a suitable way by redefining the conditional probability $p(\mathbf{x}|c)$. Other advantages are (i) the possibility of including prior class distribution $p(c)$ into the model and (ii) we can use the posterior probability $p(c|\mathbf{x})$ for classification or in other scenarios that arise in decision theory.

Section 3.3 The practical advice in this section is helpful for those who are interested in applying the classic version of NCA. The ideas are particularly useful because we are optimizing a non-convex function. We believe that using relevant component analysis (RCA; Bar-Hillel et al., 2003) for initialization and conjugate gradients for optimization works best on small and medium-sized data sets. This combination does not require tuning any parameters so it can be readily applied on any data set. Among others, we highlight the dimensionality annealing technique; this look promising: it obtained good projections that seem to avoid local optima.

Section 4.2 We gave several ideas of adapting mini-batches (MB) techniques in a suitable way for NCA method. We tested a version based on recursive projection clustering (RPC, Chalupka, 2011). The size of each mini-batch still has to be quite large to get significant gradients (we used $n = 2000$). For large data set this already represents a significant improvement in terms of cost, but we question whether there is a more principled method of selecting n, the size of a cluster.

Section 4.3 We presented a stochastic learning (SL) algorithm that is theoretically motivated by stochastic optimization arguments. Empirical investigation demonstrated that this method achieves very close scores to the classic NCA. The SL method scales with the number of the points N, but it can be further accelerated using technique presented in sections 4.4 and 4.5. A further advantage of SL is that can be used for online learning.

Section 4.4 Using the CC-KDE framework for NCA and inspired from previous work on fast KDE (Deng and Moore, 1995; Gray and Moore, 2003), we proposed an algorithm that evaluates efficiently an approximation of the NCA objective function and its corresponding gradient. This method is fastest when combined with SL method. Our experiments suggested that even if we accept a large maximum error $\epsilon_{\text{max}} = 0.1$, we do not seem to lose in accuracy. We noticed a similar behaviour when we did brute pruning.

Section 4.5 Further alterations of NCA method are the compact support version (NCA CS) and the compact support and background distribution version (NCA CS BACK). NCA CS achieves considerable speed-ups because we do not have to compute the gradient terms for all the points. Also we observe that it convergences faster than the other methods. However its speed does come at a cost: the accuracy performance is slightly worse than for the rest of the methods.

6.1 Further directions

This projected involved many possible directions. Unfortunately, in the short time allocated we could only look at a part of the whole picture and we left some open paths. We aim to continue some of the work in the near future. We plan to further investigate the automatic differentiation procedure and see whether will imply a memory blow-up or it prove to be a better idea than symbolic differentiation. Also we aim to implement a robust and efficient NCA k-d tree code. Further experimentation can be done in the dimensionality annealing idea since it seems a promising technique and we could not allocate more to it because it was not the main aim of the project.

Further refinements of our work are still possible. It would be interested to try a dual tree (Gray and Moore, 2003) inspired algorithm in the mini batch context. Also we could try this approach on other distance metric learning methods, with different objective functions (e.g., Xing et al., 2003). Although this implies losing the convexity. Who is afraid of non-convexity anyway?