17 Metric Learning

Many algorithms, and much intuition, in Data Mining start from the following set-up:

$$\text{Let } X \subset \mathbb{R}^d \text{ be a set of } n \text{ points, consider Euclidean distance } \mathtt{d}(x, x') = \|x - x'\|.$$

In reality the data encountered is much messier, and this assumption does not always hold. Consider a set of people recorded as $(\mathtt{height}, \mathtt{weight}, \mathtt{age})$. How do we measure distance between such objects?

This lecture will cover three specific scenarios.

- **Mutlidimensional Scaling** where we are given a distance $\mathtt{d}$, but it is not Euclidean. Points may not even by in $\mathbb{R}^d$.
- **Linear Discriminant Analysis** where $X \subset \mathbb{R}^d$, but we do not trust Euclidean distance. But we are given cluster labels.
- **Linear Distance Metric Learning** where $X \subset \mathbb{R}^d$, and we do not trust Euclidean distance. In this setting we are given pairs of close points, and pairs of far points.

In each case, the output will be a mapping of $\mu : X \to \mathbb{R}^k$ where we can use Euclidean distance on the resulting points. We may even be able to choose $k = 2$, and nicely visualize the data.

17.1 Multidimensional Scaling

Dimensionality reduction is an abstract problem with input of a high-dimensional data set $P \subset \mathbb{R}^d$ and a goal of finding a corresponding lower dimensional data set $Q \subset \mathbb{R}^k$, where $k << d$, and properties of P are preserved in Q. Both low-rank approximations through direct SVD and through PCA are examples of this: $Q = \pi_{V_k}(P)$. However, these techniques require an explicit representation of P to start with. In some cases, we are only presented P more abstractly. There two common situations:

- We are provided a set of n objects X, and a bivariate function $\mathtt{d} : X \times X \to \mathbb{R}$ that returns a distance between them. For instance, we can put two cities into an airline website, and it may return a dollar amount for the cheapest flight between those two cities. This dollar amount is our "distance."
- We are simply provided a matrix $D \in \mathbb{R}^{n \times n}$, where each entry $D_{i,j}$ is the distance between the ith and jth point. In the first scenario, we can calculate such a matrix D.

Multi-Dimensional Scaling (MDS) has the goal of taking such a distance matrix D for n points and giving low-dimensional (typically) Euclidean coordinates to these points so that the embedded points have similar spatial relations to that described in D. If we had some original data set A which resulted in D, we could just apply PCA to find the embedding. It is important to note, in the setting of MDS we are typically just given D, and *not* the original data A. However, as we will show next, we can derive a matrix that will act like AA^T using only D.

A *similarity matrix* M is an $n \times n$ matrix where entry $M_{i,j}$ is the similarity between the ith and the jth data point. The similarity often associated with Euclidean distance $\|a_i - a_j\|$ is the standard inner (or dot product) $\langle a_i, a_j \rangle$. We can write

$$\|a_i - a_j\|^2 = \|a_i\|^2 + \|a_j\|^2 - 2\langle a_i, a_j \rangle,$$

and hence

$$\langle a_i, a_j \rangle = \frac{1}{2}\left(\|a_i\|^2 + \|a_j\|^2 - \|a_i - a_j\|^2\right). \tag{17.1}$$

Next we observe that for the $n \times n$ matrix AA^T the entry $[AA^T]_{i,j} = \langle a_i, a_j \rangle$. So it seems hopeful we can derive AA^T from D using equation (17.1). That is we can set $\|a_i - a_j\|^2 = D_{i,j}^2$. However, we need also need values for $\|a_i\|^2$ and $\|a_j\|^2$.

Since the embedding has an arbitrary shift to it (if we add a shift vector s to *all* embedding points, then no distances change), then we can arbitrarily choose a_1 to be at the origin. Then $\|a_1\|^2 = 0$ and $\|a_j\|^2 = \|a_1 - a_j\|^2 = D_{1,j}^2$. Using this assumption and equation (17.1), we can then derive the similarity matrix AA^T. Then we can run the eigen-decomposition on AA^T and use the coordinates of each point along the first k eigenvectors to get an embedding. Taking the average of this procedure over all choices of $\|a_j\|^2 = 0$ is known as *classical MDS*.

Classical MDS. First we need to (re)define the centering matrix $C_n = I - \frac{1}{n}\mathbf{1}\mathbf{1}^T$ from PCA. Given a point set $X \subset \mathbb{R}^{n \times d}$, applying $A = C_n X$ centers X so that the mean (column-wise mean) of A is 0 in each coordinate. Recall this is the first step in PCA before calling the SVD to get the principal components and values. Its commonly written this way to encode it in linear algebra.

Now Classical MDS applies it on both sides of $D^{(2)}$ which squares all entries of D so that $D_{i,j}^{(2)} = D_{i,j}^2$. This *double centering* process creates a matrix $M = -\frac{1}{2}C_n D^{(2)} C_n$. Then the embedding uses the top k eigenvectors of M and scales them appropriately by the square-root of the eigenvalues. That is let $V\Lambda V = M$ be its eigendecomposition. The let V_k and Λ_k represent the top k eigenvectors and values respectively. The final point set is $V_k \Lambda_k^{1/2} = Q \in \mathbb{R}^{n \times k}$. So the n rows $q_1, \ldots, q_n \in \mathbb{R}^k$ are the embeddings of points to represent distance matrix D. This algorithm is sketched in Algorithm 17.1.1

Algorithm 17.1.1 Classical Multidimensional Scaling: $\mathsf{CMDS}(D, k)$

Set $M = -\frac{1}{2}C_n D^{(2)} C_n$	*# double centering*
Construct $[\Lambda, V] = \text{eig}(M)$	*# the eigendecomposition of M*
return $Q = V_k \Lambda_k^{1/2}$	*# projection to best k-dimensions*

It is often used for k as 2 or 3 so the data can be easily visualized.

There are several other forms that try to preserve the distance more directly, whereas this approach is essentially just minimizing the squared residuals of the projection from some unknown original (high-dimensional embedding). One can see that we recover the distances with no error if we use all n eigenvectors – if they exist. However, as mentioned, there may be less than n eigenvectors, or they may be associated with complex eigenvalues. So if our goal is an embedding into $k = 3$ or $k = 10$, there is no guarantee that this will work, or even what guarantees this will have. But MDS is used a lot nonetheless.

17.2 Linear Discriminant Analysis

Another tool that can be used to learn Euclidian distance for data is a *Linear Discriminant Analysis* (or LDA, or sometimes Fisher Discriminant Analysis). This term has a few variants, we focus on the multi-class setting. This means we begin with a data set $X \subset \mathbb{R}^d$, these has a known a partition of X into k classes (or clusters) $S_1, S_2, \ldots, S_k \subset X$, so $\bigcup S_i = X$ and $S_i \cap S_j = \emptyset$ for $i \neq j$.

Let $\mu_i = \frac{1}{|S_i|}\sum_{x \in S_i} x$ be the mean of class i, and let $\Sigma_i = \frac{1}{|S_i|}\sum_{x \in S_i}(x - \mu_i)(x - \mu_i)^T$ be its covariance.

Similarly, we can represent the overall mean as $\mu = \frac{1}{|X|}\sum_{x \in X} x$. Then we can then represent the *between*

class covariance as

$$\Sigma_B = \frac{1}{|X|}\sum_{i=1}^{k} |S_i|(\mu_i - \mu)(\mu_i - \mu)^T.$$

In contrast the overall *within class covariance* is

$$\Sigma_W = \frac{1}{|X|}\sum_{i=1}^{k} |S_i|\Sigma_i = \frac{1}{|X|}\sum_{i=1}^{k}\sum_{x\in S_i}(x - \mu_i)(x - \mu_i)^T.$$

The goal of LDA is a representation of X in a k'-dimensional space that maximizes the between class covariance while minimizing the within class covariance. This often formalized as finding the set of vectors u which maximize

$$\frac{u^T\Sigma_B u}{u^T\Sigma_W u}.$$

For any $k' \leq k-1$, we can directly find the orthogonal basis $U = \{u_1, u_2, \ldots, u_{k'}\}$ that maximizes the above goal with an eigen-decomposition. In particular, U is the top k' eigenvectors of $\Sigma_W^{-1}\Sigma_B$. Then to obtain the best representation of X we set the new data set as

$$\tilde{X} \leftarrow \bar{\pi}_U(X)$$

so $\tilde{x} = \bar{\pi}_U(x) = (\langle x, u_1\rangle, \langle x, u_2\rangle, \ldots, \langle x, u_{k'}\rangle) \in \mathbb{R}^{k'}$.

This retains the dimensions which show difference between the classes, and similarity among the classes. The removed dimensions will tend to show variance within classes without adding much difference between the classes. Conceptually, if the data set can be well-clustered under the k-means clustering formulation, then the U (say when rank $k' = k-1$) describes a subspace with should pass through the k centers $\{\mu_1, \mu_2, \ldots, \mu_k\}$; capturing the essential information needed to separate the centers.

17.3 Distance Metric Learning

The first approach MDS required that all distances were known ahead of time, and then a low-dimensional Euclidean embedding can be generated. The second approach LDA requires that the data X is somehow clustered or labeled into k classes before the analysis starts. In many settings these assumptions may be unrealistic.

However, if we are to choose a good metric, we must know something about which points should be close and which should be far. In the *distance metric learning* problem we assume that we have two sets of pairs; the close pairs $C \subset X \times X$ and the far pairs $F \subset X \times X$. This process starts with a dataset $X \subset \mathbb{R}^d$, and close and far pairs C and F and tries to find a metric so the close pairs are as small as possible, while the far pairs are as large as possible.

In particular, we restrict to a *Mahalanobis distance* defined with respect to a positive semidefinite matrix $M \in \mathbb{R}^{d\times d}$ on points $p, q \in \mathbb{R}^d$ as

$$\mathbf{d}_M(p, q) = \sqrt{(p-q)^T M(p-q)}.$$

So given X and sets of pairs C and F, the goal is to find M to make the close point have small $\mathbf{d}_M$ distance, and far points have large $\mathbf{d}_M$ distance. There are many ways to formulate this, and different formulations come with different algorithms.

LDA Analog

Lets start with a simple analog to LDA (for which I have not see explanation of why it works or what it explicitly optimizes). Let C have n_C pairs which we would like to be close. And let F have n_F pairs we would like to be far. Then for some small parameter $\alpha > 0$, define a $d \times d$ matrix

$$M = \left(\alpha I + \frac{1}{n_C} \sum_{\{x_i, x_i'\} \in C} (x_i - x_i')(x_i - x_i')^T - \frac{1}{n_F} \sum_{\{x_j, x_j'\} \in F} (x_j - x_j')(x_j - x_j')^T \right)^{-1}.$$

This directly defines a Mahalanobis distance $\mathbf{d}_M(p, q) = \sqrt{(p-q)^T M (p-q)}$ that tries to push close pairs together and pull far pairs apart. We can even set $M_k = V_k \Lambda_k V_k^T$ where $[\Lambda, V] = \mathsf{eig}(M)$ and Λ_k, V_k represents the top-k eigenvectors and values. Then we implicitly get a low dimensional metric.

Alterantively, we can pre-process X so that it lies in $\mathbb{R}^k$ and the distances is Euclidean. To do that decompose $M_k = A^T A$ where $A = \Lambda_k^{1/2} V_k$ and $V_k = \mathbb{R}^{k \times d}$. Now transform as

$$q_i = A x_i \quad \text{for all } x_i \in X$$

This induces that

$$\begin{aligned} \|q_i - q_j\| &= \sqrt{(Ax_i - Ax_j)^T (Ax_i - Ax_j)} \\ &= \sqrt{(x_i - x_j)^T A^T A (x_i - x_j)} \\ &= \sqrt{(x_i - x_j)^T M_k (x_i - x_j)} \\ &= \mathbf{d}_{M_k}(x_i, x_j). \end{aligned}$$

Eig-DML

In this set-up we will consider finding the optimal distance $\mathbf{d}_{M^*}$ as

$$\begin{aligned} M^* = \max_M & \min_{\{x_i, x_j\} \in F} \mathbf{d}_M(x_i, x_j)^2 \\ \text{such that} & \sum_{\{x_i, x_j\} \in C} \mathbf{d}_M(x_i, x_j)^2 \leq \kappa. \end{aligned}$$

That is we want to maximizes the closest pair in the far set F, while restricting that all pairs in the close set C have their sum of squared distances are at most κ, some constant. We will not explicitly set κ, but rather restrict M in some way so on average it does not cause much stretch. There are other reasonable similar formulations, but this one will allow for simple optimization (following Ying+Li in JMLR12).

The standard approaches in the literature set up an optimization procedure and then run a "solver" to find the best M. We will instead describe an approach which is a bit less opaque.

Notational Setup. Let $H = \sum_{\{x_i, x_j\} \in C} (x_i - x_j)(x_i - x_j)^T$; note that this is a sum of *outer* products, so H is in $\mathbb{R}^{d \times d}$. For this to work we will need to assume that H is full rank; otherwise we don't have enough close pairs to measure. Or we can set $H = H + \delta I$ for a small scalar δ.

Further, we can restrict M to have trace $\mathsf{Tr}(M) = d$, and hence satisfying some constraint on the close points. Recall that the trace of a matrix M is the sum of M's eigenvalues. Let $\mathbb{P}$ be the set of all positive semidefinite matrices with trace d; hence the identity matrix I is in $\mathbb{P}$. Also, let

$$\triangle = \{\alpha \in \mathbb{R}^{|F|} \mid \sum \alpha_i = 1 \ \& \text{ all } \alpha_i \geq 0\}.$$

Let $\tau_{i,j} \in F$ (or simply $\tau \in F$ when the indexes are not necessary) to represent a far pair $\{x_i, x_j\}$. And let $X_{\tau_{i,j}} = X_{i,j} = (x_i - x_j)(x_i - x_j)^T \in \mathbb{R}^{d \times d}$, an outer product. Let $\tilde{X}_\tau = H^{-1/2} X_\tau H^{-1/2}$. It turns out our optimization goal is now equivalent (up to scaling factors, depending on κ) to finding

$$\arg\max_{M \in \mathbb{P}} \min_{\alpha \in \triangle} \sum_{\tau \in F} \alpha_\tau \langle \tilde{X}_\tau, M \rangle.$$

Here $\langle X, M \rangle = \sum_{s,t} X_{s,t} M_{s,t}$, a dot product over matrices, but because since X will be related to an outer product between two data points, this makes sense to think of as $\mathbf{d}_M(X)$.

Optimization procedure. Given the formulation above, we will basically try to find an M which stretches the far points as much as possible while keeping $M \in \mathbb{P}$. We do so using a general procedure referred to as Frank-Wolfe optimization, which increases our solution using one data point (in this case a far pair) at a time.

Set $\sigma = d \cdot 10^{-5}$ as a small smoothing parameter. Define a gradient as

$$g_\sigma(M) = \frac{\sum_{\tau \in F} \exp(-\langle \tilde{X}_\tau, M \rangle / \sigma) \tilde{X}_\tau}{\sum_{\tau \in F} \exp(-\langle \tilde{X}_\tau, M \rangle / \sigma)}.$$

Observe this is a weighted average over the $\tilde{X}_\tau$ matrices. Let $v_{\sigma,M}$ be the maximal eigenvector of $g_\sigma(M)$; the direction of maximal gradient.

Then the algorithm is simple. Initialize $M_0 \in \mathbb{P}$ arbitrarily; for instance as $M_0 = I$. Then repeatedly find for $t = 1, 2, \ldots$ as (1) find $v_t = v_{\mu, M_{t-1}}$, and (2) set $M_t = \frac{t-1}{t} M_{t-1} + \frac{1}{t} v_t v_t^T$. This is summarized in Algorithm 17.3.1.

Algorithm 17.3.1 Optimization for DML

Initialize $M_0 = I$.
for $t = 1, 2, \ldots, T$ **do**
  Set $G = g_\sigma(M_{t-1})$
  Let $v_t = v_{\sigma, M_{t-1}}$; the maximal eigenvalue of G.
  Update $M_t = \frac{t-1}{t} M_{t-1} + \frac{1}{t} v_t v_t^T$.
return $M = M_T$.