Locally Linear Embedding

Locally linear embedding (LLE), is an unsupervised learning algorithm that computes low-dimensional neighborhood preserving embeddings.

- Given $N$ real-valued vectors $X_i$, each of dimensionality $D$.
- Suppose data point and its neighbors lie on or close to a locally linear patch.
- Characterize the local geometry of these patches by linear coefficients that reconstruct each data point from its neighbors.

Reconstruction errors are measured by cost function

$$E(W) = \sum_i (X_i - \sum_j W_{ij}X_j)^2.$$
Locally Linear Embedding (cont.)

• Cost function adds up squared distances between all the data points and their reconstructions.

• Weights $W_{ij}$ summarize the contribution of the $j$th data point to the $i$th reconstruction.

• For determining the weights $W_{ij}$ the cost function is minimized subject to two constraints.
  1. Data point $X_i$ is reconstructed only from its neighbors (enforcing $W_{ij} = 0$ if $X_j$ does not belong to the set of neighbors of $X_i$).
  2. Rows in the weight matrix sum to 1: $\sum_j W_{ij} = 1$.

• Optimal weights $W_{ij}$ subject to these constraints are found by solving a least-squares problem.
Locally Linear Embedding (cont.)

- Constrained weights that minimize the reconstruction errors obey an important symmetry
  - For any particular data point they are invariant to rotations, rescalings, and translations of that data point and its neighbors.

In the final step of LLE, high-dimensional observation $X_i$ is mapped to a low-dimensional vector $Y_i$ by choosing $d$-dimensional coordinates $Y_i (d \ll D)$ such that the embedding cost function

$$\Phi(Y) = \sum_i (Y_i - \sum_j W_{ij} Y_j)^2.$$

is minimized. However, now we fix the weights $W_{ij}$ while optimizing the coordinates $Y_i$. 
Nonlinear dimensionality reduction by locally linear embedding.
Method of Steepest Descent

Let $E(w)$ be a continuously differentiable function of some unknown (weight) vector $w$.

Find an optimal solution $w^*$ that satisfies the condition

$$E(w^*) \leq E(w).$$

The necessary condition for optimality is

$$\nabla E(w^*) = 0.$$

Let us consider the following iterative descent:

Start with an initial guess $w^{(0)}$ and generate sequence of weight vectors $w^{(1)}, w^{(2)}, \ldots$ such that

$$E(w^{(i+1)}) \leq E(w^{(i)}).$$
Steepest Descent Algorithm

\[ w^{(i+1)} = w^{(i)} - \eta \nabla E(w^{(i)}) \]

where \( \eta \) is a positive constant called learning rate.

At each iteration step the algorithm applies the correction

\[ \Delta w^{(i)} = w^{(i+1)} - w^{(i)} = -\eta \nabla E(w^{(i)}) \]

Steepest descent algorithm satisfies:

\[ E(w^{(i+1)}) \leq E(w^{(i)}) \]

to see this, use first-order Taylor expansion around \( w^{(i)} \) to approximate \( E(w^{(i+1)}) \) as \( E(w^{(i)}) + (\nabla E(w^{(i)}))^T \Delta w^{(i)} \).
Steepest Descent Algorithm (cont.)

$$E(w^{(i+1)}) \approx E(w^{(i)}) + (\nabla E(w^{(i)}))^T \Delta w^{(i)}$$

$$= E(w^{(i)}) - \eta \|\nabla E(w^{(i)})\|^2$$

For positive learning rate $\eta$, $E(w^{(i)})$ decreases in each iteration step (for small enough learning rates).

$$3x^2 + y \exp(-x^2 - y^2)$$
Steepest Descent Algorithm Example

Black points denote different starting values. Learning rate $\eta$ is properly chosen, however for starting value $(1, 1)$, algorithm converges not to the global minimum. It follows steepest descent in the “wrong direction”, in other words, gradient based algorithms are local search algorithms.

$$z = (3x_1^2 + x_2) \exp(-x_1^2 - x_2^2)$$

$\eta = 0.25$
Learning rate $\eta = 1.0$ is too large, algorithm oscillates in a "zig-zag" manner or "overleap" the global minimum.
Steepest Descent Algorithm Example (cont.)

Learning rate $\eta = 0.005$ is too small, algorithm converges “very slowly”.

$$z = (3x_1^2 + x_2) \exp(-x_1^2 - x_2^2)$$

$\eta = 0.005$
Single-Layer Network

Equivalent notation:

\[ y(x) = \tilde{w}^T \tilde{x} = \sum_{i=0}^{d} w_i x_i \]

where \( \tilde{w} = (w_0, w) \) and \( \tilde{x} = (1, x) \).
Linear Classifier

- Linear classifiers are single layer neural networks.

Observe, that $x_2 = 2x_1$ can also be expressed as

$$w_1 x_1 + w_2 x_2 = 0 \iff x_2 = -\frac{w_1}{w_2} x_1,$$

where for instance

$$w_1 = -2, \ w_2 = 1.$$ 

Furthermore, observe that all points lying on the line $x_2 = 2x_1$ satisfy $w_1 x_1 + w_2 x_2 = -2x_1 + 1x_2 = 0.$
What about the vector \( w = (w_1, w_2) = (-2, 1) \)?

Vector \( w \) is perpendicular to the line \(-2x_1 + 1x_2 = 0\).

Let us calculate the dot product of \( w \) and \( x \).

The dot product is defined as
\[
w_1x_1 + w_2x_2 + \ldots + w_dx_d \overset{def}{=} \langle w, x \rangle, \text{ for some } d \in \mathbb{N}.
\]

In our example \( d = 2 \) and we obtain \(-2 \cdot 1 + 1 \cdot 2 = 0\).
Let us consider the weight vector $\mathbf{w} = (3, 0)$ and vector $\mathbf{x} = (2, 2)$.

Geometric interpretation of the dot product: Length of the projection of $\mathbf{x}$ onto the unit vector $\mathbf{w}/\|\mathbf{w}\|$. 

\[
\langle \mathbf{w}, \mathbf{x} \rangle \|\mathbf{w}\| = \frac{3 \cdot 2 + 0 \cdot 2}{\sqrt{3^2}} = 2
\]
Linear Classifier & Two Half-Spaces

The $x$-space is separated in two half-spaces.

\[ \{ x | -2x_1 + 1x_2 > 0 \} \]

\[ \{ x | -2x_1 + 1x_2 < 0 \} \]

\[ \{ x | -2x_1 + 1x_2 = 0 \} \]
• Observe, that \( w_1 x_1 + w_2 x_2 = 0 \) implies, that the separating line always goes through the origin.

• By adding an offset (bias), that is

\[
w_0 + w_1 x_1 + w_2 x_2 = 0 \iff x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2} \equiv y = mx + b,
\]

one can shift the line arbitrary.
Note that $x_0 = 1$, $y(x) = \langle w, x \rangle + w_0$.

Given data which we want to separate, that is, a sample $\mathcal{X} = \{(x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N)\} \in \mathbb{R}^d \times \{-1, +1\}$.

How to determine the proper values of $w$ such that the “minus” and “plus” points are separated by $y(x)$? Infer the values of $w$ from the data by some learning algorithm.
Perceptron

Note, so far we have not seen a method for finding the weight vector $w$ to obtain a linearly separation of the training set.

Let $g(a)$ be (sign) activation function

$$g(a) = \begin{cases} 
-1 & \text{if } a < 0 \\
+1 & \text{if } a \geq 0 
\end{cases}$$

and decision function

$$g(\langle w, x \rangle) = g \left( \sum_{i=0}^{d} w_i x_i \right).$$

Note: $x_0$ is set to $+1$, that is, $x = (1, x_1, \ldots, x_d)$. Training pattern consists of $(x, t) \in \mathbb{R}^{d+1} \times \{-1, +1\}$.
Perceptron Learning Algorithm

**input**: $(x_1, t_1), \ldots, (x_N, t_N) \in \mathbb{R}^{d+1} \times \{-1, +1\}$, $\eta \in \mathbb{R}_+$, $\text{max.epoch} \in \mathbb{N}$

**output**: $w$

begin
    Randomly initialize $w$
    epoch $\leftarrow 0$

repeat
    for $i \leftarrow 1$ to $N$ do
        if $t_i \langle w, x_i \rangle \leq 0$ then
            $w \leftarrow w + \eta x_i t_i$
        epoch $\leftarrow$ epoch + 1
    until (epoch $=$ max.epoch) or (no change in $w$)
return $w$
end

– p. 117
Training the Perceptron (cont.)

Geometrical explanation: If $x$ belongs to $\{+1\}$ and $\langle w, x \rangle < 0 \Rightarrow$ angle between $x$ and $w$ is greater than $90^\circ$, rotate $w$ in direction of $x$ to bring misclassified $x$ into the positive half space defined by $w$. Same idea if $x$ belongs to $\{-1\}$ and $\langle w, x \rangle \geq 0$. 
Perceptron Error Reduction

Recall: missclassification results in:

\[ w_{\text{new}} = w + \eta x_t, \]

this reduces the error since

\[-w_{\text{new}}(x_t)^T = -w(x_t)^T - \eta \left( x_t (x_t)^T \right)_{>0} \left( x_t \right)_{>0} \left\| x_t \right\|^2_{>0} \]

\[ < -w^T x_t \]

How often one has to cycle through the patterns in the training set?

• A finite number of steps?
**Proposition 1** Given a finite and linearly separable training set. The perceptron converges after some finite steps [Rosenblatt, 1962].
Perceptron Algorithm (R-code)

```r
perceptron <- function(w, X, t, eta, max.epoch) {
    N <- nrow(X)/2;
    epoch <- 0;
    repeat {
        w.old <- w;
        for (i in 1:(2*N)) {
            if (t[i]*y(X[i,],w) <= 0)
                w <- w + eta * t[i] * X[i,];
        }
        epoch <- epoch + 1;
        if (identical(w.old,w) || epoch = max.epoch) {
            break; # terminate if no change in weights or max.epoch reached
        }
    }
    return (w);
}
```
One epoch terminate if no change in $w$