Learning Graphs for Semi-supervised Learning

Using Leave-one-out Self-consistency

Part I. Notations

We use the harmonic energy minimization algorithm (Zhu 2003). Assume there are \( L \) labeled data and \( U \) unlabeled data. \( N = L + U \). The weight matrix \( W \) is defined as:

\[
\begin{pmatrix}
W_{LL} & W_{LU} \\
W_{UL} & W_{UU}
\end{pmatrix}
\]

with each element being non-negative. Define \( p_{ij} = w_{ij} / \sum_{k=1}^{N} w_{ik} \) so \( P = (p_{ij}) \) is not necessarily symmetric. Let the label of the \( L \) labeled data be \( f_L \).

Then using Zhu’s harmonic energy minimization model, we have \( f'_{U} = (I - P_{UU})^{-1} P_{UL} f_L \). This \( f_{U} \) can be interpreted from the perspective of random walk along the graph \( G \). Starting from an unlabeled node \( i \), it moves to a node \( j \) with probability \( p_{ij} \) after one step. The walk continues until the particle hits a labeled node. Then the \( i^{th} \) element of \( f_{U} \) is the probability that the particle, starting from node \( i \), hits a labeled node with label 1 (in contrast to 0). Here the labeled data is viewed as an “absorbing boundary” for the random walk.

Part II. Formulation

Now for LOO hyper-parameter (graph) learning, we examine the case when the \( t^{th} \) labeled data is left out. Define \( P'_{UU} = \begin{pmatrix} p_{ii} & P_{iU} \\ P_{Uj} & P_{UU} \end{pmatrix}, \) where \( P_{Li} = (p_{Li+1,1}, ..., p_{Li,N}) \). Define \( \begin{pmatrix} p_{i1} & \cdots & p_{i,t-1} & p_{i,t+1} & \cdots & p_{iL} \\ p_{iL+1,1} & \cdots & p_{iL+1,t-1} & p_{iL+1,t+1} & \cdots & p_{iL+1,L} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ p_{N,1} & \cdots & p_{N,t-1} & p_{N,t+1} & \cdots & p_{N,L} \end{pmatrix} \), \( (t = 1 \ldots L) \). Then the LOO prediction on the \( t^{th} \) labeled example is \( f'_{U} = (I - P'_{UU})^{-1} P'_{UL} f'_L \), where \( f'_L \) is the vector after removing the \( t^{th} \) element of \( f_L \).

So the objective function to be maximized is: \( Q = \sum_{t=1}^{L} c_t s^T (I - P'_{UU})^{-1} P'_{UL} f'_L \), where \( c_t \)
is the class of the \( t \)th example and \((U+1)\times1\) vector \( s = (1, 0, ..., 0)^T\). Note \( c_t \) takes value in \(-1/+1\) while the elements of \( f^*_L \) take value in 0/1 respectively.

Assuming \( w_{ij} = \exp \left( -\sum_{d=1}^F \left( x_{ij} - x_{j,d} \right)^2 \right) / \sigma_d^2 \), then the partial derivatives are:

\[
\frac{\partial Q}{\partial \sigma_d} = \sum_{t=1}^L c_t s^T \left( \left( I - P^t_{UU} \right)^{-1} \frac{\partial P^t_{UU}}{\partial \sigma_d} \left( I - P^t_{UU} \right)^{-1} P^t_{UL} f^*_L + \left( I - P^t_{UU} \right)^{-1} \frac{\partial P^t_{UL}}{\partial \sigma_d} f^*_t \right)
\]

\[
= \sum_{t=1}^L c_t s^T \left( I - P^t_{UU} \right)^{-1} \left( \frac{\partial P^t_{UL}}{\partial \sigma_d} f^*_L + \frac{\partial P^t_{UL}}{\partial \sigma_d} f^*_t \right)
\]

where the fact that \( dX^{-1} = -X^{-1} \left( dX \right) X^{-1} \) is applied.

Then \( \frac{\partial P^t_{UL}}{\partial \sigma_d} \) and \( \frac{\partial P^t_{UL}}{\partial \sigma_d} \) are submatrices of \( \frac{\partial P}{\partial \sigma_d} \). Since \( p_{ij} = w_{ij} / \sum_{k=1}^N w_{ik} \), we have

\[
\frac{\partial p_{ij}}{\partial \sigma_d} = \frac{\partial w_{ij}}{\partial \sigma_d} - p_{ij} \sum_{k=1}^{L+U} \frac{\partial w_{ik}}{\partial \sigma_d} \frac{\partial w_{ik}}{\partial \sigma_d}
\]

Finally, \( \frac{\partial w_{ij}}{\partial \sigma_d} = 2w_{ij} \left( x_{ij} - x_{j,d} \right)^2 / \sigma_d^3 \).

**Part III. Computational Complexity, a Naive Algorithm**

It is easy to verify that the inverse of \( I - P^t_{UU} \) can be calculated efficiently for all \( t = 1...L \) by applying matrix inversion lemma. However, we still need to calculate the inverse once in each iteration, after all \( \sigma \)’s are updated in the last iteration.

The following is the pseudo-code that naively calculates the gradient in LOO-Hyperparameter learning:

Function value: \( F_n = 0 \), Gradient: \( \mathbf{G} = (0, \ldots, 0)^T \) (\( \mathbf{G} \) is a \( F \)-dimensional vector).

For each \( t \in 1...L \) (Leave-one-out loop)

\[
P^t_{UU} \leftarrow \left( \begin{array}{c|c} p_{Ut} & p_{UL} \\ \hline p_{UL} & P^t_{UU} \end{array} \right), \quad p_{Ut} = (p_{L+1}, \ldots, p_{N,t})^T \quad \text{and} \quad p_{UL} = (p_{t,L+1}, \ldots, p_{t,N})
\]

\[
P^t_{UL} \leftarrow \left( \begin{array}{cccc} p_{L1} & \cdots & p_{L,t-1} & p_{L,t+1} & \cdots & p_{L} \\ p_{L+1,t-1} & \cdots & p_{L+1,t-1} & p_{L+1,t+1} & \cdots & p_{L+1} \\ \vdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ p_{N,t-1} & \cdots & p_{N,t-1} & p_{N,t+1} & \cdots & p_{N,L} \end{array} \right)
\]
\[ f_U^t \leftarrow \left( I - P_{UU}^t \right)^{-1} P_{UL}^t f_L^t. \]  

(3)  

\[ F_n \leftarrow F_n + c_i s^T \left( I - P_{UU}^t \right)^{-1} P_{UL}^t f_L^t. \]  

(4)  

\( W_{UU}^t \) and \( W_{UL}^t \) be the corresponding weight matrix, using the same index as in \( P_{UU}^t \) and \( P_{UL}^t \). We will denote as \( w_{UU}^t(i,j) \) the \((i, j)^{th}\) element of \( W_{UU}^t \). Although there are only \( U \) and \( L \) columns in \( W_{UU}^t \) and \( W_{UL}^t \) respectively, we will use \( \sum_{k=1}^{N} \frac{\partial w_{UU}^t(i,k)}{\partial \sigma_d} \) to denote the sum of the row in the original \( W \), which corresponds to the \( i^{th} \) row of \( W_{UU}^t \). Actually, for \( i \geq 2 \) the corresponding row doesn’t change for different \( t \) (an important property that we will utilize later). The first row \((i=1)\) always corresponds to the labeled data point currently left out.

For each \( d \) in \([1...F]\)  

(for all features) \[ \frac{\partial P_{UU}^t(i,j)}{\partial \sigma_d} = \frac{1}{\sum_{k=1}^{N} w_{UU}^t(i,k)} \left( \frac{\partial w_{UU}^t(i,j)}{\partial \sigma_d} - p_{UU}^t(i,j) \sum_{k=1}^{N} \frac{\partial w_{UU}^t(i,k)}{\partial \sigma_d} \right) \]  

(5)  

\[ \quad i \in [1...U+1], j \in [1...U+1] \]

\[ \frac{\partial P_{UL}^t(i,j)}{\partial \sigma_d} = \frac{1}{\sum_{k=1}^{N} w_{UL}^t(i,k)} \left( \frac{\partial w_{UL}^t(i,j)}{\partial \sigma_d} - p_{UL}^t(i,j) \sum_{k=1}^{N} \frac{\partial w_{UL}^t(i,k)}{\partial \sigma_d} \right) \]  

(6)  

\[ \quad i \in [1...U+1], j \in [1...L-1] \]

where \( \frac{\partial w_{UL}^t(i,j)}{\partial \sigma_d} \), if corresponds to the \( p^{th} \) and \( q^{th} \) example (index of the original \( W \)), equals \( 2 w_{pq} \left( x_{p,d} - x_{q,d} \right)^2 / \sigma_d^3 \).

\[ G_d \leftarrow G_d + c_i s^T \left( I - P_{UU}^t \right)^{-1} \left( \frac{\partial P_{UU}^t(\cdot, \cdot)}{\partial \sigma_d} f_U^t + \frac{\partial P_{UL}^t(\cdot, \cdot)}{\partial \sigma_d} f_L^t \right) \]  

(7)  

where \( \frac{\partial P_{UU}^t(\cdot, \cdot)}{\partial \sigma_d} \) stands for the matrix composed of all \( \frac{\partial P_{UU}^t(i,j)}{\partial \sigma_d} \) for \( i, j \in [1...U+1] \). Likewise \( \frac{\partial P_{UL}^t(\cdot, \cdot)}{\partial \sigma_d} \).

End
The naive algorithm is very computational expensive: $O(U^2 FL)$, just for one iteration of the gradient descent. This is due to following drawbacks. First, it ignores the fact that for each example $x_i$, there may be only a small number of features whose value is nonzero (active features). This is particularly true for text datasets. Secondly, in the Leave-one-out loop, a lot of variables are repeatedly calculated which constitutes a huge waste. However, it requires significant amount of care to extract the common parts out of the loop in order to reduce the complexity to $O\left(U^2 \max(U, F_m)\right)$ for calculating the gradient, where $F_m$ is the maximum number of active features among all examples. This is already the lower bound because it is inevitable to calculate $\frac{\partial W_{ij}}{\partial \sigma_d} = 2 w_{ij} \left(x_{id} - x_{jd}\right)^2 / \sigma_d^3$ (for $i, j \in 1..U, d \in 1..F_m$), particularly $\left(x_{id} - x_{jd}\right)^2$ which can not be further factorized. Although $\left(x_{id} - x_{jd}\right)^2$ is constant across all iterations, we can not pre-compute it because the cost for storing the result is $O(U^2 F_m)$, usually beyond the memory capacity. Perhaps it is fine to pre-compute part of it to make best use of memory available. The following describes the factorization procedure which yields $O(U^2 F_m)$ for gradient calculation.

**Part IV. Tackling Computational Complexity, an Efficient Algorithm**

The first costly step is matrix inversion, which is $O\left(U^3\right)$. Using Matlab, this cost is no longer the bottleneck. The second expensive step is in (7), particularly the partial derivatives:

$$\frac{\partial P'_{UL}(\cdot, \cdot)}{\partial \sigma_d} f_U' + \frac{\partial P'_{UL}(\cdot, \cdot)}{\partial \sigma_d} f_L'$$

$$\frac{\partial P'_{UL}(i, j)}{\partial \sigma_d} = \frac{1}{\sum_{k=1}^N w_{UL}^i (i, k)} \left( \frac{\partial w_{UL}^i (i, j)}{\partial \sigma_d} - \frac{p_{UL}^i (i, j)}{\partial \sigma_d} \right)$$

$$\frac{\partial P'_{UL}(i, j)}{\partial \sigma_d} = \frac{1}{\sum_{k=1}^N w_{UL}^i (i, k)} \left( \frac{\partial w_{UL}^i (i, j)}{\partial \sigma_d} - \frac{p_{UL}^i (i, j)}{\partial \sigma_d} \right)$$

Examining them carefully,
\[
\frac{\partial P'_{UU}(i,\cdot)}{\partial \sigma_d} f'_U = \frac{1}{\sum_{k=1}^{N} w'_{UU}(i,k)} \left( \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} - p'_{UU}(i,\cdot) \sum_{k=1}^{N} \frac{\partial w'_{UU}(i,k)}{\partial \sigma_d} \right) f'_U \\
= \frac{1}{\sum_{k=1}^{N} w'_{UU}(i,k)} \left( \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} f'_U - p'_{UU}(i,\cdot) f'_U \cdot \sum_{k=1}^{N} \frac{\partial w'_{UU}(i,k)}{\partial \sigma_d} \right) \\
\]  

(8)

where \( \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} \) and \( p'_{UU}(i,\cdot) \) are interpreted as row vectors covering all proper indices for the “.”. Here we have already made use of an important property of \( P'_{UU} \), namely each row corresponds to the transfer property from a fixed graph node to other nodes. See (1) where all elements in each row have the same first subscript. Otherwise the equation in (8) (lifting \( \sum_{k=1}^{N} \frac{\partial w'_{UU}(i,k)}{\partial \sigma_d} \) out as common factor) does not necessarily hold. Note \( P'_{UU} \) also has such property which will also be utilized.

Looking at computational complexity, it costs \( O\left(N^2 F_m\right) \) to calculate the \( W \) and \( P \).

\[
\sum_{k=1}^{N} w'_{UU}(i,k) \text{ costs } O(N) \text{ for a given } i, \quad O(UN) \text{ to cover all } i, \text{ and } \quad O(UNL) \text{ to cover all } t. \quad \sum_{k=1}^{N} \frac{\partial w'_{UU}(i,k)}{\partial \sigma_d} \text{ costs } O(NF_m) \text{ for a given } (i, t) \text{ (with all the } F_m \text{ features already covered), and } \quad O(UNF_m) \text{ to cover all } i. \quad \text{Since only the first row and first column of } W'_{UU} \text{ changes for different } t, \text{ it costs } O(N^2 F_m) \text{ to cover all } t. \quad p'_{UU}(i,\cdot)f'_U \text{ costs } O(U) \text{ (so } O(U^2) \text{ to cover all } i \text{ and } O(U^2L) \text{ for all } t). \quad \text{The only term left is } \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} f'_U, \text{ which costs } O(UF_m) \text{ (so } O(U^2 F_m) \text{ to cover all } i \text{ and } O(U^2 LF_m) \text{ to cover all } t). \quad \text{This will be too costly for just calculating the gradient. Fortunately, we noticed that with } t \text{ varied, } \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} \text{ is basically constant: only the first row and first column change with } t. \quad \text{So we only need to record the } f'_U \text{ for all } t \text{ and then traverse all items in } \frac{\partial w'_{UU}(i,\cdot)}{\partial \sigma_d} \text{ only once (in contrast to } L \text{ times).}
In this way the cost to cover all \( i \) and \( t \) is \( O(UNF_m) \). In sum, the costs are:

\[
\frac{\partial P^i_{UL}(\cdot, \cdot)}{\partial \sigma_d} f_U^i = \frac{1}{\sum_{k=1}^N w_{UL}^i(i,k)} \left( \frac{\partial w_{UL}^i(i, \cdot)}{\partial \sigma_d} f_U^i - p_{UL}^i(i, \cdot) f_U^i \cdot \sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} \right)
\]

with \( i, t \) fixed to cover all \( i \) and to cover all \( t \).

\begin{align*}
\begin{array}{c|c|c|c}
\text{w., } P & \text{with } i, t \text{ fixed} & \text{to cover all } i & \text{to cover all } t \\
\sum_{k=1}^N w_{UL}^i(i,k) & O\left(N^2 F_m\right) & O\left(N^2 F_m\right) & O\left(N^2 F_m\right) \\
\sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} & O\left(N F_m\right) & O\left(N F_m\right) & O\left(N F_m\right) \\
\frac{\partial p_{UL}^i(i, \cdot)}{\partial \sigma_d} f_U^i & O\left(U\right) & O\left(U^2\right) & O\left(U^2 L\right) \\
\frac{\partial w_{UL}^i(i, \cdot)}{\partial \sigma_d} f_U^i & O\left(UNF_m\right) & O\left(UNF_m\right) & O\left(UNF_m\right)
\end{array}
\end{align*}

Secondly,

\[
\frac{\partial P_{UL'}^i(\cdot, \cdot)}{\partial \sigma_d} f_{UL'}^i = \frac{1}{\sum_{k=1}^N w_{UL}^i(i,k)} \left( \frac{\partial w_{UL}^i(i, j)}{\partial \sigma_d} f_{UL'}^i - \frac{\partial p_{UL}^i(i, j)}{\partial \sigma_d} f_{UL'}^i \cdot \sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} \right)
\]

\begin{align*}
\sum_{k=1}^N w_{UL}^i(i,k) \text{ costs } O(UN) \text{ to cover all } i \text{ and } O(UNL) \text{ to cover all } t.
\sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} \text{ is exactly the same as in calculating } (8). \quad p_{UL}^i(i, \cdot) f_{UL'}^i \text{ costs } O(L),
\end{align*}

so \( O(UL) \) to cover all \( i \) and \( O(UL^2) \) to cover all \( t \). \( \frac{\partial w_{UL}^i(i, \cdot)}{\partial \sigma_d} f_{UL'}^i \) costs \( O(LF_m) \),

so \( O(ULF_m) \) to cover all \( i \) and \( O(UL^2 F_m) \) to cover all \( t \). Like in calculating (8), it is also possible to make use of the fact that different \( t \) only results in different first row and two different columns. In sum, the costs are:

\[
\frac{1}{\sum_{k=1}^N w_{UL}^i(i,k)} \left( \frac{\partial w_{UL}^i(i, \cdot)}{\partial \sigma_d} f_{UL'}^i - \frac{\partial p_{UL}^i(i, \cdot)}{\partial \sigma_d} f_{UL'}^i \cdot \sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} \right)
\]

with \( i, t \) fixed to cover all \( i \) and to cover all \( t \).

\begin{align*}
\begin{array}{c|c|c|c}
\text{w., } P & \text{with } i, t \text{ fixed} & \text{to cover all } i & \text{to cover all } t \\
\sum_{k=1}^N w_{UL}^i(i,k) & O\left(N^2 F_m\right) & O\left(N^2 F_m\right) & O\left(N^2 F_m\right) \\
\sum_{k=1}^N \frac{\partial w_{UL}^i(i,k)}{\partial \sigma_d} & O\left(N F_m\right) & O\left(N F_m\right) & O\left(N F_m\right) \\
\frac{\partial p_{UL}^i(i, \cdot)}{\partial \sigma_d} f_{UL'}^i & O\left(U\right) & O\left(U^2\right) & O\left(U^2 L\right) \\
\frac{\partial w_{UL}^i(i, \cdot)}{\partial \sigma_d} f_{UL'}^i & O\left(UNF_m\right) & O\left(UNF_m\right) & O\left(UNF_m\right)
\end{array}
\end{align*}
\[ \sum_{k=1}^{N} w_{i,k} = O(N) \quad \text{or} \quad O(UN) \quad \text{or} \quad O(UNL) \]

\[ \sum_{k=1}^{N} \frac{\partial w_{i,k}}{\partial \sigma_d} = O(\text{NF}_m) \quad \text{or} \quad O(UNF_m) \quad \text{or} \quad O(N^2 F_m) \]

\[ p'_{UL}(i,j) f'_{L} = O(L) \quad \text{or} \quad O(UL) \quad \text{or} \quad O(UL^2) \]

\[ \frac{\partial w '_{L,i}}{\partial \sigma_d} f'_L = O(LF_m) \quad \text{or} \quad O(ULF_m) \quad \text{or} \quad O(UL^2 F_m) \]

The memory cost is \( O(N^2) \), just to store a fixed number of matrices. In my implementation, I only used some \( N \times F \) matrices.

**Part V. Parallelization**

A simple way to apply parallel processing is through dividing features. Since all expensive calculations (underlined in above tables) involve \( F_m \), it improves performance to distribute the partial derivatives to different cluster nodes. Note the parallelism for calculating \( W \) and \( P \) is different from the other partial derivatives and parallelism is needed only for calculating the exponent (summation for RBF). \( W \) and \( P \) are always calculated in the first place with all other calculations depending on it. Here we must make use of the fact that each example may only have a small number of active features. Each processor calculates the contribution of their features to the exponent of edge weight between \( x_i \) and \( x_j \). Finally the master processor assembles those \( N(N+1)/2 \)-sized (upper triangle) tables to calculate \( W \). Using parallelism for calculating the partial derivatives is similar. In all, parallel processing can reduce complexity to \( O(N^2 F_m/P) \), where \( P \) is the number of processors available.

Furthermore, how to divide the features into several sets is also a problem, if we want the load on different processors to be balanced. Though this is an NP-hard problem, we only need an approximate solution. Suppose we have a list of length \( F \), each element recording \( a_f = \sum_i \sum_j \text{cover}(x_i, x_j, f) \). \( \text{cover}(x_i, x_j, f) \) is 0 if neither \( x_i \) nor \( x_j \) assumes nonzero value on feature \( f \). Otherwise, it equals 1. So the problem is:

given \( N \) examples, \( F \) features, and \( P \) processors, find a partition of \( C = \{1, 2, \ldots, F\} \):

\( A_1, \ldots, A_P \) (\( \bigcup_{k=1}^{P} A_k = C \)), \( A_i \cap A_j = \emptyset \) for \( \forall i \neq j \), such that the maximum of

\[ \text{load}(k) = \sum_{f \in A_k} a_f = \sum_{f \in A_k} \sum_i \sum_j \text{cross}(x_i, x_j, f) \]

is as small as possible. For simplicity, we only need to consider unlabeled \( x_i \) and \( x_j \), because the leave-one-out loop treats
only one labeled data as unlabeled. Therefore, this sub-optimal partition needs to be computed only once given the (labeled and) unlabeled examples.