CS 6241: Numerics for Data Science

Function approximation on graphs

David Bindel

2025-04-10

Semi-supervised learning

Suppose we have a collection of objects that we want to classify one of two ways. Given some labeled examples, how should we label the remaining objects? This is a standard *semi-supervised learning* task. Of course, labels alone do not help us unless we have some idea how the objects are related to each other. In this lecture, we will assume that this information comes in the form of a weighted graph, where the objects to be classified are vertices and the edge weights represent the degree of similarity or connectedness. Our problem, then, is to label the remaining objects so that — as much as possible — similar objects will share the same label.

Before writing methods, we will first introduce some notation. We will start with the two-class case, and turn to the multi-class problem later. Let x be the vector of class labels; ideally, we would like $x \in \{0,1\}^n$. We order the vertices so that the labeled examples appear last, and partition x into unlabeled and labeled subvectors:

$$x = \begin{bmatrix} u \\ y \end{bmatrix},$$

where $u \in \{0,1\}^{n_u}$ is unknown and $y \in \{0,1\}^{n_y}$ is known. We let the weighted adjacency matrix A encode the similarity, and let $L = D - A$ be the weighted Laplacian. To measure the quality of a class assignment, we look at the quadratic

$$x^T L x = \sum_{(i,j) \in \mathcal{E}} a_{ij} (x_i - x_j)^2$$

which, for 0-1 vectors, gives the total weight of all between-class edges. We partition L and A conformally with the partitioning of x:

$$L = \begin{bmatrix} L_{uu} & L_{uy} \\ L_{yu} & L_{yy} \end{bmatrix}.$$

Then we have
$$x^T L x = u^T L_{uu} u + 2u^T L_{uy} y + y^T L_{yy} y.$$
Alas, optimizing this function with respect to the class assignments u is a challenging discrete optimization problem.

Soft labels

The optimization is easier if we relax the problem, replacing binary class labels with real-valued soft labels. Then we have a continuous quadratic optimization for which the critical point equation is
$$[Lx]_u = L_{uu} u + L_{uy} y = 0.$$
The matrix L is positive semi-definite, with null vectors that are constant on each connected component; but we will assume that we have at least one labeled example in each connected component, so that L_{uu} is nonsingular.

It is worth looking at the scalar equations in order to understand this system in more detail. Let us write row i of the critical point equations as
$$d_i x_i - \sum_{j=1}^n a_{ij} x_j = 0,$$
and rearrange to find
$$x_i = \sum_{j=1}^n \frac{a_{ij}}{d_i} x_j.$$
The weights a_{ij}/d_i are non-negative and sum to one, so this tells us that for each i where the label is unknown, we are choosing x_i to be the *weighted average* of the neighbor labels. This tells us that (for example) all of the computed soft labels will be in the interval $[0, 1]$.

The averaging interpretation of the equlibrium of the equation suggests an algorithm for computing the soft labels by interpreting the averaging operation as an update equation:
$$x_i^{\mathrm{new}} = \sum_{j=1}^n \frac{a_{ij}}{d_i} x_j.$$

Several classical *point relaxation* methods of numerical linear algebra follow this approach, differing in the order in which the updates are computed and applied. Jacobi iteration updates the entire u vector based on the old guesses; Gauss-Seidel sweeps through the labels and updates them in a fixed order, using the most recent guesses in each update; and Gauss-Southwell chooses the next label to update adaptively, based on the size of a corresponding residual element. In machine learning, these are known as *label propagation* methods, though label propagation methods generally include an additional rounding operation to turn soft labels into hard labels at each step. The convergence of such iterations depends strongly on the nature of the similarity graph: if it is “tightly connected,” the iterations converge quickly, while the iterations may converge much more slowly if the connectivity is relatively sparse. We will return to this point later.

From Laplacians to kernels

We have seen an expression of the form

$$u = -L_{uu}^{-1} L_{uy} y$$

once before, in our initial discussion of Gaussian processes. There, instead of the graph Laplacian, we saw the *precision matrix* (the inverse of the covariance). We would therefore *like* to say that L^{-1} is a kernel. Of course, we have to worry about a slight caveat: L is not invertible! Hence, while we can still define a kernel associated with L, we will have to use a conditionally positive definite kernel associated with the *pseudo-inverse* $L^\dagger$.

The pseudoinverse as a kernel

The Laplacian pseudo-inverse is $L^\dagger$, corresponding to the minimal-norm least-squares solution to linear systems with L. In terms of the eigendecomposition $L = Q \Lambda Q^T$, the pseudo-inverse is $L^\dagger = Q \Lambda^\dagger Q^T$ where $\lambda_i^\dagger = \lambda_i^{-1}$ for nonzero λ_i, and is zero otherwise. Note that $LL^\dagger = L^\dagger L = J$ where J is the centering matrix $J = I - ee^T/n$.

Indeed, we can think of the soft label problem as a kernel method involving the (conditionally positive definite) kernel matrix $L^\dagger$; that is,

$$u = [L^\dagger]_{uy} c + \mu e$$

where the weight vector c is given by

$$\begin{bmatrix} [L^\dagger]_{yy} & e \\ e^T & 0 \end{bmatrix} \begin{bmatrix} c \\ \mu \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix}.$$

To see this is equivalent to what we wrote before, we observe that

$$\begin{aligned} L_{uu}[L^\dagger]_{uy} + L_{uy}[L^\dagger]_{yy} &= J_{uy} = ee^T/n \\ L_{uu}e + L_{uy}e &= 0 \end{aligned}$$

Because $e^T c = 0$ by construction, we therefore have

$$L_{uu}u + L_{uy}y = L_{uu}([L^\dagger]_{uy}c + \mu e) + L_{uy}([L^\dagger]_{yy}c + \mu e) = 0,$$

which is indeed the equation that we used to define u previously.

Laplacian features

It is also helpful to think about this kernel in terms of feature vectors. Let $\Psi^T = Q' \Lambda'^{-1/2}$, where Q' and Λ' are the parts of the eigendecomposition corresponding to the nonzero eigenvalue, so that $L^\dagger = \Psi^T \Psi$. The columns of Ψ are the feature vectors in the graph associated with the kernel, and the soft label function is equivalent to $x_i = \psi_i^T d + \mu$ where d is the minimal norm vector such that $\Psi_y^T d + \mu e = y$. To see that this is equivalent, consider the constrained optimization

$$\text{minimize } \frac{1}{2}\|d\|^2 \text{ s.t. } \Psi_y^T d + \mu e = y,$$

and note that the KKT equations are

$$\begin{bmatrix} I & \Psi_y & 0 \\ \Psi_y^T & 0 & e \\ 0 & e^T & 0 \end{bmatrix} \begin{bmatrix} d \\ \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} 0 \\ y \\ 0 \end{bmatrix}.$$

Eliminating the first equation $d = -\Psi_y \lambda$ gives us

$$\begin{bmatrix} -\Psi_y^T \Psi_y & e \\ e^T & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \mu \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix},$$

which we can rewrite as

$$\begin{bmatrix} [L^\dagger]_{yy} & e \\ e^T & 0 \end{bmatrix} \begin{bmatrix} -\lambda \\ \mu \end{bmatrix} = \begin{bmatrix} y \\ 0 \end{bmatrix}.$$

This is the same system that we saw a moment ago, but with $c = -\lambda$ reinterpreted as a vector of Lagrange multipliers. Therefore, the minimal norm coefficient vector in the feature space is $d = \Psi_y c$, which gives us the prediction

$$u = \Psi_u^T d + \mu e = \Psi_u^T \Psi_y c + \mu e = [L^\dagger]_{uy} c + \mu e.$$

We will see the eigenvector features associated with the $L^\dagger$ kernel again next time when we address *unsupervised* learning with graphs.

Electrical analogies

So far, we have focused on a purely mathematical intuition for the soft labeling problem. But we can also consider a more physical picture. We will consider the flow of current through a resistor network, which is a common choice in this business[1] We suppose there are n nodes connected by resistors. At each node, we have a voltage v_i, and on each resistor edge we have a resistance r_{ij}. There are two basic ingredients to the equations:

[1]Other analogies involve pressure-driven flow through a pipe network or motion of a spring network.

- A *constitutive law*: For a linear resistor, the current from i to j is
$$I_{ij} = r_{ij}^{-1}(v_i - v_j).$$
- A *balance law*: The total current leaving a node is zero, or
$$\sum_j I_{ij} = 0.$$

Putting these two ingredients together gives us the system
$$\sum_{j \in N_i} r_{ij}^{-1}(v_i - v_j) = 0$$
at each node i for which we do not explicitly control the voltage (by attaching the node to ground or a voltage supply) or inject a current. This gives us a weighted Laplacian linear system, where the Laplacian is known as the *conductance matrix* in circuit theory, and the edge weights a_{ij} are the element conductances (inverse resistances[2]). Hence, the soft labeling problem is equivalent to drawing a resistive circuit network and attaching some nodes to a unit voltage supply (the examples labeled 1) and others attached to ground (the examples labeled 0). The intuition is that nodes that are connected by low-resistance edges or paths tend to have similar voltages. The Laplacian quadratic form is associated with resistive power loss.

Whether the analogy to circuit theory provides insight or not probably depends on your background. But the analogy is sufficiently widely used that it is worth knowing about, whether or not you find it provides you with any personal intuition.

Kernels and distances

Positive definite kernels define inner products in a feature space, and inner products define a Euclidean distance structure. That is, if ψ is a feature map for a kernel on a space $\mathcal{X}$, then
$$\begin{aligned} \|\psi(x) - \psi(y)\|^2 &= \psi(x)^T\psi(x) - 2\psi(x)^T\psi(y) + \psi(y)^T\psi(y) \\ &= k(x,x) - 2k(x,y) + k(y,y). \end{aligned}$$

In the positive definite case, we can therefore use the kernel to define a squared distance on $\mathcal{X}$:
$$d(x,y)^2 = k(x,x) - 2k(x,y) + k(y,y),$$
and this distance satisfies all the properties that a distance is supposed to satisfy (positivity, symmetry, and the triangle inequality).

[2]In a circuit theory class, I would write the conductances as $g_{ij} = r_{ij}^{-1}$. But to maintain notational consistency with the rest of the lecture, we will use a_{ij} here.

Of course, the kernel associated with the graph Laplacian is only positive *semi*-definite because of the null vector. The usual hazard for semi-definite kernel functions is that we might have distinct points in $\mathcal{X}$ with the same feature vector, and a distance between two points is supposed to be nonzero if the points are distinct. We do not have to worry about this problem with the Laplacian kernel, though, as the construction in this case looks like

$$d_{ij}^2 = (e_i - e_j)^T L^\dagger (e_i - e_j);$$

and since the vectors $e_i - e_j$ are orthogonal to the null vector of all ones, this quantity will be positive for all $i \neq j$.

We sometimes call d_{ij}^2 the *resistance distance*, since in the electrical analogy it corresponds to the effective resistance between nodes i and j summarized over all possible network paths. In the physical analogy, the current balance law holds in the following generalized sense: if S is the set of nodes for which we have specified voltages (label information), then for any $i \notin S$,

$$\sum_{j \in S} d_{ij}^{-2} (v_i - v_j) = 0;$$

we can rewrite this as

$$v_i = \frac{\sum_{j \in S} d_{ij}^{-2} v_j}{\sum_{j \in S} d_{ij}^{-2}};$$

that is, the computed value at node i is a weighted average of the known values, where the weights are proportional to the inverse-square distances. This formula for the soft labeling function works even with other kernel functions — though, of course, we lose the circuit analogy!

The heat kernel

So far, we have focused on the inverse Laplacian graph kernel. However, this is not the only choice! Another kernel that we can use for many of the same purposes is the *heat kernel*, which is given by $\exp(-tL)$. The parameter is associated with time, and the entries of $\exp(-tL)$ can be interpreted in terms of the diffusion of heat from a source at i to a target at j within time t. Alternately, the entries $\exp(-tL)_{ij}$ can be interpreted as the probability that a continuous random walk starting from i will be at j at time t.

Extending to multi-class learning

So far, we have focused on the two-class case with 0-1 labels. For the more general case where we want k different classes, we use the same technique applied to k indicator vectors, one for each class. That is, we replace the vector $x \in \mathbb{R}^n$ with the matrix $X \in \mathbb{R}^{n \times k}$. In the hard label case, we let x_{ik} be one if i belongs to class k and zero otherwise. In the soft label case, we assign node i to the class k for which x_{ik} is maximal. We also have that $\sum_k x_{ik} = 1$, and so sometimes x_{ik} is interpreted as the probability that node i belongs to class k.

The Laplace solver building block

We conclude this lecture with a brief discussion of the landscape of methods for solving Laplacian linear systems.

For small systems — up to a few thousand nodes — there is not much to discuss. In these cases, forming and factoring the Laplacian matrix as a dense matrix is usually fine, and requires little thought or care. Past a few thousand nodes, though, the $O(n^3)$ cost of a dense matrix factorization becomes prohibitive. In this case, we can either

- Use a *sparse direct* method that computes a factorization in less than $O(n^3)$ time, or
- Use an iterative solver.

Of course, the two methods are not mutually exclusive, and we often use approximate factorizations as *preconditioners* for iterative methods. But it is important to recognize that many graphs are *either* well suited to iterative methods *or* well suited to sparse direct solvers. The key distinction is whether the graph can be separated by relatively small cuts (a problem we will consider in the next lecture).

When a graph can be partitioned with a small cut, we can try to solve it by a *divide and conquer* approach. Suppose that there is a small *vertex separator* that partitions the graph into two roughly-equal size pieces. If we label the two separate pieces first and then put the separator at the end, then we can write the Laplacian system in block form as

$$L = \begin{bmatrix} L_{11} & 0 & L_{13} \\ 0 & L_{22} & L_{23} \\ L_{31} & L_{32} & L_{33} \end{bmatrix}.$$

The structure comes from the observation that the degrees of freedom in the two pieces (block 1 and block 2) are not directly connected. Block Gaussian elimination on the system gives us

$$\begin{aligned} S &= L_{33} - L_{31}L_{11}^{-1}L_{13} - L_{32}L_{22}^{-1}L_{23} \\ Sx_3 &= b_3 - L_{31}L_{11}^{-1}b_1 - L_{32}L_{22}^{-1}b_2 \\ L_{22}x_2 &= b_2 - L_{23}x_3 \\ L_{11}x_1 &= b_1 - L_{13}x_3 \end{aligned}$$

Hence, if we can quickly solve systems with L_{11} and L_{22}, then we can form and solve a much smaller *Schur complement* system to couple them together. The *nested dissection* approach applies this idea recursively, and gives us a very fast solver *if we can find small separators.*

Of course, the extreme case of small separators is when we have a tree. In this case we can produce very fast solvers that run in linear time in the matrix size. One way to see this is in the electrical network analogy: we can compute the resistance between any pair of nodes quickly because it is just the sum of the resistances along the unique path between those nodes! More generally, graphs that are associated with nearest neighbor connectivity in 2D (or sometimes 3D) tend to have small *tree width*, and are good for sparse solvers. There are good sparse solvers in the world, and I do not recommend writing your own. But it is important to know what graphs are well suited to sparse solvers.

The opposite extreme is when there are no small separators. In this case, though, the smallest nonzero eigenvalue of the Laplacian is usually far from zero, so that the condition number of the Laplacian system is not too large. This is exactly the situation in which standard iterative methods work well.