Garbage In, Garbage Out: Learning the right graph

Adam G. Erickson
adam.erickson@yale.edu

"On two occasions I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question."
— Charles Babbage, *Passages from the Life of a Philosopher*

Abstract

Many recent machine learning algorithms have taken advantage of an underlying graph (or network) structure to solve tasks more effectively. Sometimes the graph structure is a given prior in the problem, but oftentimes this is not the case. This paper focuses on the later situation, where a graph representation is not given but would be helpful. The graph must be learned from the data. Frequently this is done with simple and intuitive methods, like a k-nearest-neighbors graph or a Gaussian radial basis function to weight the edges. In this work we study more general constructions for the graph, motivated by the growing field of Graph Signal Processing. We examine how these methods perform on some synthetic examples constructed from a known underlying graph, and some real world datasets of biological phenomena where a ground-truth graph is not as readily available. Comparisons are made to the prior art of graph construction, as well as some connections with probabilistic graphical models (PGMs). These learned graphs are then used as input for a wide variety of machine learning algorithms, with the hope that an improved graph input improves the performance of downstream analysis.

1 Introduction

1.1 Problem Setup

In machine learning the most common representation of a dataset is a matrix. (Depending on your subfield) the rows are viewed as samples of some object under consideration, like cells, people, movies, images, etc., and the columns are the features of aspects of each row that are measured, such as expression of specific genes, height and weight attributes, review score, certain pixels, etc. This gives us a canonical $X \in \mathbb{R}^{n \times d}$ object to analyze.

Oftentimes the data has an underlying graph structure intrinsic to the problem. Things like sensor networks, temperature distributions, social network graphs, contagion networks in epidemiology, etc. provide good examples. When these problems come with a well defined graph, we can exploit the underlying structure of the graph in order to improve our model of the problem. In recent years, the developing field of Graph Signal Processing has appeared to simplify and enhance the study of data (signals) on such a graph. (Also, recent interest

in geometric deep learning and graph convolutional networks has piqued interest in graph representations of data).

However, sometimes we are not given a graph. What are we to do then? Are graph methods of any use to us? In fact, there are several popular machine learning methods that exploit graph structure when no obvious graph is involved in the model of the problem. Most of the time these methods take the form of constructing a similarity graph between the samples. That is, treat each row as a vertex, and then figure out what the edges should be between each vertex. Once such a graph is obtained, there are multiple methods which attempt to make conclusions about the original dataset by doing analysis on this graph.

Obviously, the "quality" of the constructed graph will impact the quality of the final results. We want these graphs to have nice properties, like non-negative edge weights, a single connected component, no self-loops, undirected edges. Oftentimes simple methods with some parameter tuning are used to construct the graph, as far more interest is paid to the algorithm analyzing the graph. We think that the prevalence of these graphs in a large set of methods means any incremental improvement in their construction could have far-reaching effects. So how can we better construct these graphs?

1.2 Background

Some assumptions and background relevant to the problem.

Spectral Graph Theory A lot of the theory presented here rests on the fact that you can describe most everything with associated matrices. This turns modern graph theory into a lot of advanced linear algebra. As explored in the problem set forwards and course lecture notes (so I will not reproduce here in full), the Laplacian matrix defined by:

$$L = D - W$$

where D is the degree matrix and W our weighted adjacency matrix for the graph. The laplacian is positie semi-definite, and when we have a function defined on the nodes of the graph we often like to consider the laplacian quadratic form, which looks like:

$$f^T L f = \frac{1}{2} \sum_{i,j=1} w_{ij}(f_i - f_j)^2$$

this form of the laplacian can be used as a regularized, and its been explored many places like Belkin and Niyogi (2001), Zhu, Ghahramani, Lafferty, et al. (2003).

Manifold Assumption In many problem applications we assume that our data has some structure; frequently, we assume that our data is sampled from a manifold with dimension less than the number of dimensions we are measuring. Then we have a lower-dimensional manifold embedded in a high d dimensional space. This often makes intuitive sense: if you are hoping to find something underlying your dataset, then you have it imagine it has some lower dimensional structure than that which you are observing. For example, with images that are taken as you rotate some object, like a tank or person. We know a human observer could describe each image with a high degree of accuracy using only a single degree of freedom (the angle of rotation). That is, there is some structure in a 1-dimensional space underlying what

an average human being observer is seeing. The second (probably obvious?) assumption is that the human being is not privy to any sort of information that a camera could not capture to determine the answer to this problem. The human can tell the single parameter, degree of rotation, just by looking at photographs, which of course a computer or camera could "look at" as well. Ergo, this 1-dimensional underlying structure must be present in the photographic information.

Graphs approximate manifolds Since a manifold is a continuous object, we cannot hope to collect it as data, because data collection can only sample numbers with finite precision. So, to discretize the problem some we can view a graph constructed on the data samples as a discrete approximation of a continuous manifold. Operators that are defined in continuous mathematics, like the laplacian, can then be viewed as analogues of similar operators on graphs, like the graph laplacian matrix. Going the other way, you can also view manifolds as continuous approximations to graphs. Note the mapping is not bijective. See [1] for more on the theoretical framework.

1.3 Common Graph Constructions

Criteria Without being too mathematically rigorous, we often want certain properties for the graph we construct on our dataset. To make it easy to work with, we desire non-negative edge weights, undirected, and no-self loops. Usually we want the graph to be connected (except sometimes in clustering). There is a trade off between this (connectivity) and our other typical desire that the graph be sparse. If we have a dense graph we don't learn too much, because each node is more or less a friend of every other node. If we have a too-sparse graph that is disconnected, well we don't have enough edges to even have a single graph. My helpful intuition is that graphs that approximate meshes would be nice, as well as in the extreme case planar graphs.

KNN Perhaps the simplest construction to imagine. For each vertex x, we find the other k closest points to it, and connect them with an edge. Do this for every node. Note this graph is not symmetric, like a mutual nearest neighbors graph might be (beyond the scope of this work).

Gaussian We use a Gaussian kernel to construct a graph by making each weight w_{ij} between node i and j inverse proportional to the euclidean distance between the points.

$$w_{ij} = \exp\left(- \frac{\|x_i - x_j\|_2^2}{2\sigma^2} \right) \tag{1}$$

Adaptive Kernel In order to correct for the relative density around (sampled) points, we can form another version of the kernel like:

$$\kappa(x_i, x_j) = \frac{1}{2} \exp\left(- \frac{\|x_i - x_j\|_2^2}{2\sigma_k(x_i)^2} \right) + \frac{1}{2} \exp\left(- \frac{\|x_i - x_j\|_2^2}{2\sigma_k(x_j)^2} \right)$$

where $\sigma_k(x)$ is the distance from x to its k-th nearest neighbor.

Probabilistic Graphical Models We'll stay away from these today because they can be more general, and admit things like directed edges and negative edge weights. Still, many of these methods are important in network inference.

1.4 How GSP methods differ

In contrast to the prevailing methods for studying graph creation, several GSP methods consider global properties of the graph, instead of just pairwise or neighborhood properties. In the example of k-nearest-neighbors or an epsilon ball, you are making one point the "protagonist" and then creating a neighborhood around it, looping through to do this for each point. In contrast, when we examine global properties of the graph, or signals over the graph like the Laplacian quadratic form, we can construct an optimization problem where we vary parameters on the global structure of the graph. While both methods may have their advantages, this allows us to find graph configurations that take into account constraints not possible in the local, "greedy" approach.

1.5 State of the art

Today we'll compare several common constructions with two recent methods in graph inference from smooth signals. In [2] Dong et al. addressed the graph learning problem by adopting a factor analysis model for graph signals and imposing a Gaussian probabilistic prior on the independent variables which model the signal. Their results showed that smoothness property of a graph signal is favoured by the efficient representation obtained through these priors. In particular, in the algorithm they presented they deployed the use of l^1 and Frobenius norm, the former expressed by a constraint on the Trace of the Laplacian, and eventually accounts for a sparsity term, while the latter is added as a penalty term in the objective function in order to control the distribution of the off-diagonal entries in L.

At the same time, in [3] Kalofolias proposes another framework to learn a graph structure under the smoothness assumption. They also use the minimization of the trace term for the Laplacian matrix, claiming that the minimizations of it brings to naturally sparse solutions. We will examine [3] and [2] in depth as we implement some Python code to solve the problem of graph learning, and compare it to the standard approach.

2 GSP Constructions

2.1 Setup for GSP Graph Construction

We consider our typical dataset like

$$X \in \mathbb{R}^{n \times d}$$

where we can consider the rows of the matrix like vector samples, such that

$$X = [x_1 \dots x_n]^T$$

and we assume that each x_i is a node on some graph G. This means the number of nodes of G is $|G(V)| = n$.

Now, for the signal processing perspective. Since we have this graph on n nodes, and each column of X is a vector defined on every node, we view each column of X as a signal on

this graph of n nodes. That is, each column of X is n-dimensional, and there are exactly d columns that we have at our disposal. The justification for treating the rows as nodes and the columns as signals appears to me, for the time being, to be one of convenience. Indeed, our prior assumption is just that an underlying graph exists. If one had a clever way of assuming that there exits a graph with all of the columns (features) as nodes, you could take the transpose of X and repeat the same procedure. We may examine experiments of this form further down. The only practical consideration this imposes on you is that oftentimes $n \gg d$, and this might have implications for the time and space complexity of your algorithm. It also may impact the assumptions of just how much data you have to learn this graph. But of course, there are exceptions to every rule, and based on your dataset and prior assumptions you may build a graph on either the rows or the columns. (If you can, why not both?!)

2.2 The smoothness assumption

OK, so now we have some row vectors that are nodes, and columns that are signals on this graph. What does this help us do? Well, it helps us form some priors on the structure of the graph. Taking a step back, one of the common simple assumptions about data residing on graphs is that the signal changes smoothly between connected nodes. A simple way to quantify how smooth a set of vectors on a weighted undirected graph is through the function:

$$\frac{1}{2}\sum_{i,j} W_{i,j}||x_i - x_j||^2 \tag{2}$$

with W_{ij} representing the weight between node i and node j, as usual, and x_i and x_j representing two vectors in $x_1, x_2, \ldots, x_N \in \mathbb{R}^d$. This means that if two vectors x_i and x_j are two nodes connected by a high W_{ij}, then they are expected to have a small distance; on the other hand, if two nodes are considered to have a big distance, then the weight of the edge connecting them should be small.

Comparison Lets compare the smoothness assumption to the classical signal processing domain with just a single index, in order to develop some simple intuition. Say we are sampling audio of clean pure tones. We have a bunch of different notes, each note is a different sample. Then each "vertex" of the graph here would be a certain time index. And the x_i at that vertex would be the collections of samples evaluated at the time index i. Then, another x_j is the "vertex" at a different time point. If these two vertices are close in time, we expect their values to be close, since fore each sample we do no expect it to change very much in the limit as the two time indices i and j approach each other. This is not making the claim that the samples are differential, which is a tricky thorn in stochastic processes without Ito's calculus, but merely that the change from nearby indicies is expected to be small.

2.3 Solving for Smoothness

Laplacian Formulation .

Our above constraint can be re-expressed using the graph laplacian like:

$$tr(X^T L X) \tag{3}$$

Many methods exist that use the above constraint on the laplacian to regularize the signals in things like matrix completion, in a general form like:

$$\min_X \ [g(X) + tr(X^T L X)] \tag{4}$$

see for example: Zheng et al. (2011) for graph regularized sparse coding.

From our descriptions above, in order to learn the underlying graph what we are really interested in is a sort of sister problem:

$$\min_{L \in \mathcal{L}} \ [g(L) + tr(X^T L X)] \tag{5}$$

where $L \in \mathcal{L}$ means we are looking for a laplacian matrix out of the set of valid possibilities for a laplacian matrix, that satisfy the properties of such a matrix.

Adjacency Matrix Representations .

Above we have an optimization problem on the Laplacian. We know that the laplacian matrix uniquely defines a graph, and that there is a one to one relationship between the laplacian, the weighted adjacency matrix, and the graph under consideration. So optimizing over the laplacian is the same as optimizing over the weight matrix W. If we define a pairwise distance matrix with the letter Z, so we don't get confused with the degree matrix, as:

$$Z_{i,j} = \|x_i - x_j\|^2$$

then using the graph weighted adjacency matrix W we can rewrite the above optimization as:

$$tr(X^T L X) = \frac{1}{2} tr(WZ) = \frac{1}{2} \|W \circ Z\|_{1,1} \tag{6}$$

where the last term is really an element-wise product of the two matrices, just like you could get in numpy. It is referred to as the "hadamard product". The subscript on the norm tells us to take the element-wise 1-norm of the resulting matrix inside. This gives an interesting interpretation: **The term above for smoothness of signals on a graph is a weighted l-1 norm of** W. Based on the usual use and interpretation of the $l-1$ norm, this term encodes a **weighted form of sparsity**, so it will serve as a sparsity term that more heavily penalizes non-sparsity in *distant rows of* X. If the distances between the rows are sampled from a smooth manifold, then the corresponding graph to approximate the manifold should be sparse, and only form edges in small local neighborhoods. This is a common idea in current graph construction methods.

2.4 Additional Terms

We want an objective function of the form (now in terms of the weighted adjacency matrix)

$$\min_{W \in \mathcal{W}} \quad g(W) + \|W \circ Z\|_{1,1} \tag{7}$$

what shall we choose for our additional terms?

Log Barrier A barrier function is a way of replacing an inequality constraint in an optimization problem, and it is sort of a differentiable relaxation of an indicator function. Basically, in a minimization problem we have a term that goes to infinity at the boundary of the feasible set, so it is a penalty that avoids travelling outside the feasible region. It falls into a class of interior point methods as it keeps us in the interior of the solid defined by the optimization problem. Here, we use a log barrier function on the degree vector of all the nodes. Since $\log(0)$ is undefined, and $\lim_{x\to 0}\log(x) = -\infty$, a negative log penalty helps bound the degree away from 0. This is good, because we want to avoid the trivial solution of no edges, which means the laplacian quadratic form is minimized. To have a graph we need edges! This gives us a form like:

$$\min_{W\in\mathcal{W}} \quad \|W \circ Z\|_{1,1} - \alpha \mathbf{1}^T \log(W\mathbf{1}) \tag{8}$$

This is a good penalty, because it means that degrees are forced to be positive and each node has at least one edge. Of course this does not guarantee connectivity, but it is a step in that direction. It does this while still managing our **sparsity / connectivity tradeoff**, because even though the degree of each node has to be positive, it still lets any arbitrary edge go to zero without violating this constraint.

Frobenius Norm Because the above would probably lead to very sparse graphs, we think to add another term as well that penalizes the formation of very big weights, but not small ones as much. This can help us with creating local neighborhoods, but if we scale this term too much it can produce very dense graphs, where we'd prefer sparse (and connected). Adding the parameter gives us:

$$\min_{W\in\mathcal{W}} \quad \|W \circ Z\|_{1,1} - \alpha \mathbf{1}^T \log(W\mathbf{1}) + \beta\|W\|_F^2 \tag{9}$$

and now we've got our objective function!

2.5 Alternative Setups: Dong et al.

Prior to the above construction first suggested by Kalofolias 2016, Dong et. al. provided a procedure for learning the laplacian of the graph from assumptions that the sampled data were smooth signals. In fact, Dong proposed a two step process that alternated back and forth between learning the graph laplacian, and learning a new representation of the input signal that was smoother on the graph. In Dong's formulation, this took the form of two equations:

$$\begin{aligned} \underset{L}{\text{minimize}} \quad & \alpha \operatorname{tr}(Y^T L Y) + \beta\|L\|_F^2 \\ \text{s.t.} \quad & \operatorname{tr}(L) = n \end{aligned} \tag{10}$$

as well as learning a smoothed version of the signal, without compromising the reconstruction loss to the original data too much:

$$\min_{Y} \quad \|X - Y\|_F^2 + \alpha \operatorname{tr}(Y^T L Y) \tag{11}$$

This is also an interesting approach, but we believe might be slightly inferior to the above derivation. The addition of the back and forth signal smoothing / graph learning is certainly interesting.

Since the laplacian smoothing term was recast in our setup above, let us also formulate the problem from Dong in a similar fashion. We see the parameter s is scale of the problema and α controls the scale by weighting the frobenius norm. Recast using the adjacency matrix, this formulation can be seen as:

$$\begin{aligned} \min_{W \in \mathcal{W}} \quad & \|W \circ Z\|_{1,1} + \alpha \|W\mathbf{1}\|^2 + \alpha \|W\|_F^2 \\ \text{s.t.} \quad & \|W\|_{1,1} = s \end{aligned} \tag{12}$$

.

Which appears as a more comparable form. So now we've got some optimization problems. Let's create some solvers!

3 Methods

First, all of the common graph construction methods were implemented. Any details on the common constructions or their parameter selection, are left for the experiment section. This section deals with solving the optimization problems that we are presenting as new methods for comparison, from the section above.

3.1 Algorithm Description

To solve the optimization problem we use methods that work well on the general class of problems formulated like:

$$\min_{W \in \mathcal{W}} \quad g(W) + \|W \circ Z\|_{1,1}$$

of which we are considering the two special cases described above. The optimization literature is a little "thick" for me here, and most of the numerical schemes are already implemented in packages that would be silly to reinvent the wheel. I've done my best to break it down into fundamental pieces below.

Primal Dual Methods For every optimization problem (the primal) there is another problem (the dual) where finding a solution for one is equivalent to finding a solution for another. There are rigorous mathematical notions of strong and weak duality, but at least for here the intuition is in a game theoretic sense: one of the problems is your strategy, and the dual is the opponents strategy. If the primal is a minimization with constraints, the dual is typically a different "perspective" for looking at the problem that takes the form of a maximization of a different function with different constraints. However, the dual can always be mechanically generated from the primal. Primal dual methods take advantage of both of these problems to iteratively solve more complicated optimization problems.

Operator splitting As a very general idea, operator splitting is a lot like divide and conquer, parallel computations, separation of variables, etc. They originated for solving things like differential equations. More recently, splitting methods have been adapted to solve composite optimization problems, which involve sums of convex functions composed with linear operators, by breaking them down into simple sub-problems. So, we are going to want to break up our objective function into "separable" pieces, and then solve them separately.

Proximity Operator Roughly, say f is some function $f : \mathbb{R}^N \to \mathbb{R}$. Now, for every vector in the input space $x \in \mathbb{R}^N$, we can define a convex optimization problem like:

$$\underset{y \in \mathbb{R}^N}{\operatorname{argmin}} \quad f(y) + \frac{1}{2}\|x - y\|^2 \tag{13}$$

There is a unique solution for every x. So we denote the solution as $\text{prox}_f(x)$, and we know $\text{prox}_f(x) : \mathbb{R}^N \to \mathbb{R}^N$. Here is the intuition: this is very similar to a denoising problem or to a projection operator onto some other space. We want to minimized the *reconstruction* between x and y, while also having some additional functional penalty for y.

Primal Dual Proximal Splitting Putting it all together, we want to split our objective function up into a sum of different sub problems, concurrently solve the primal and dual problem in a clever way (this may help lead to the splitting in the first place), and for any non-differentialble terms that appear we can use the proximity operator. (Differentialble terms we can use a gradient operator scheme).

For more on these methods in general (because this subject can fill it's own paper), see "An Overview of Recent Primal-Dual Approaches for Solving Large-Scale Optimization Problems" by Nikos Komodakis. A popular algorithm in deep learning that involves a lot of these concepts is the ADMM algorithm.

3.2 Specifics

So for our general problem we first switch to a **vector of weights** instead of the symmetric weight matrix, just so we don't have to worry about the symmetry constraint and the complications of dealing with the matrix. We look to obtain a splitting like:

$$\min_{w \in \mathcal{W}_v} \quad f_1(w) + f_w(Kw) + f_3(w)$$

where K is a linear operator. We find nice forms like this for the preceding methods we discussed. Then, these forms plug nicely into **optimization packages** available in many languages like matlab and python. The rest is handeled for us by these computational methods.

3.2.1 GSP1

The split three functions we obtain for the longer derivation above, where z is the vectorized form of the distance matrix, is:

which we will refer to as **GSP-1**

$$
\begin{aligned}
f_1(w) &= \mathbb{1}\{w \geq 0\} + 2w^\top z, \\
f_2(d) &= -\alpha \mathbf{1}^\top \log(d), \\
f_3(w) &= \beta \|w\|^2, \text{ with } \zeta = 2\beta,
\end{aligned}
$$

GSP1

3.2.2 GSP2

For the work proposed by Dong we have:

$$
\begin{aligned}
f_1(w) &= \mathbb{1}\{w \geq 0\} + 2w^\top z, \\
f_2(c) &= \mathbb{1}\{c = s\}, \\
f_3(w) &= \alpha \left(2\|w\|^2 + \|Sw\|^2\right), \text{ with } \zeta = 2\alpha(m+1),
\end{aligned}
$$

GSP2

which we will refer to as **GSP-2**

3.3 Pseudocode

And if you really want to see the Pseudocode, much of which is abstracted away by python packages, below it is in all its glory.

4 Experiments and Results

4.1 Synthetic Model

Of course we start with a synthetic model where we can construct a ground truth graph, and see how each of our various methods approaches the problem of learning the graph. Roughly we,

1. Construct an artificial graph (ground truth)
2. Generate (smooth) signals on the graph
3. Choose a graph construction method
4. That method uses the signals to infer the graph
5. Compare the learned graph the the ground truth graph
6. Compare learned graph to other approximations

and we try to do this on a couple models.

Algorithm 1 Primal dual algorithm for model (12).

Input: z, α, β, $w^0 \in \mathcal{W}_v$, $d^0 \in \mathbb{R}^m_+$, γ, tolerance ϵ
for $i = 1, \ldots, i_{max}$ **do**
  $y^i = w^i - \gamma(2\beta w^i + S^\top d^i)$
  $\bar{y}^i = d^i + \gamma(S w^i)$
  $p^i = \max(0, y^i - 2\gamma z)$
  $\bar{p}^i = (\bar{y}^i - \sqrt{(\bar{y}^i)^2 + 4\alpha\gamma})/2$ ▷ elementwise
  $q^i = p^i - \gamma(2\beta p^i + S^\top p^i)$
  $\bar{q}^i = \bar{p}^i + \gamma(S p^i)$
  $w^i = w^i - y^i + p^i$;
  $d^i = d^i - \bar{y}^i + \bar{q}^i$;
  if $\|w^i - w^{i-1}\|/\|w^{i-1}\| < \epsilon$ **and**
    $\|d^i - d^{i-1}\|/\|d^{i-1}\| < \epsilon$ **then**
    break
  end if
end for

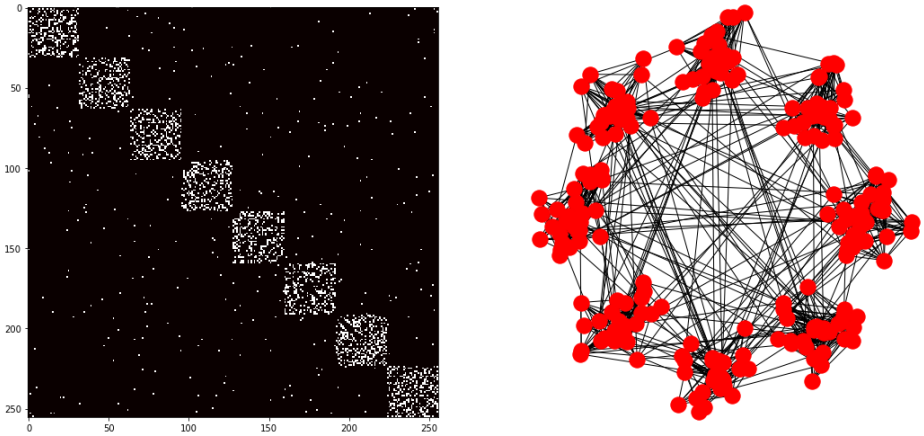
GSP-1

Algorithm 2 Primal dual algorithm for model Dong et al. 2015a.

Input: $z, \alpha, s, w^0 \in \mathcal{W}_v$, $c^0 \in \mathbb{R}_+$, γ, tolerance ϵ
for $i = 1, \ldots, i_{max}$ **do**
  $y^i = w^i - \gamma(2\alpha(2w^i + S^\top S w^i) + 2c^i)$
  $\bar{y}^i = c^i + \gamma(2\sum_j w^i_j)$
  $p^i = \max(0, y^i - 2\gamma z)$
  $\bar{p}^i = \bar{y}^i - \gamma s$
  $q^i = p^i - \gamma(2\alpha(2p^i + S^\top S p^i) + 2p^i)$
  $\bar{q}^i = \bar{p}^i + \gamma(2\sum_j p^i_j)$
  $w^i = w^i - y^i + q^i$;
  $c^i = c^i - \bar{y}^i + \bar{q}^i$;
  if $\|w^i - w^{i-1}\|/\|w^{i-1}\| < \epsilon$ **and**
    $|c^i - c^{i-1}|/|c^{i-1}| < \epsilon$ **then**
    break
  end if
end for

GSP-2

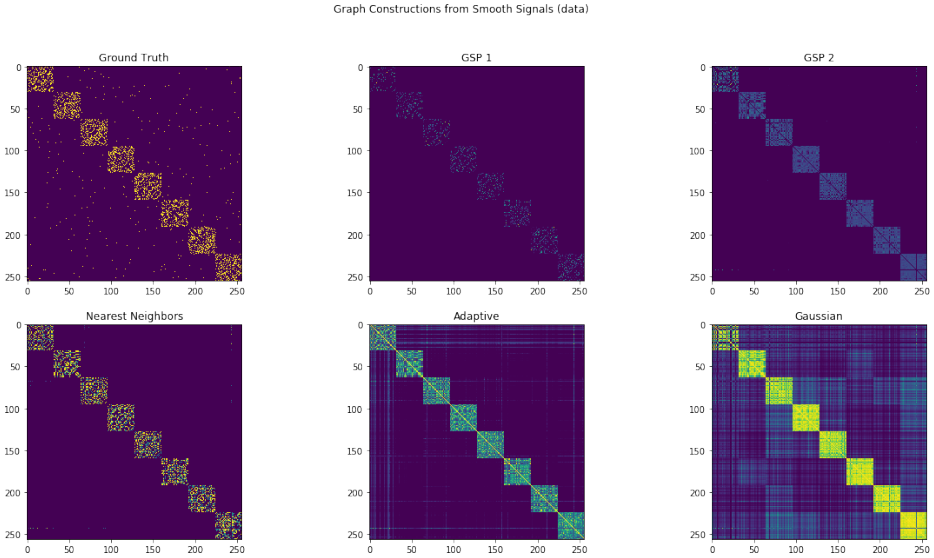
Stochastic Block model, N=256, Nc=8, p=0.5, q=0.07



4.1.1 Stochastic Block Model

Ah, our old friend, the SBM. A good illustration of breaking eigenvalue degeneracies with perturbation, and a good generative model for graph clustering. We generate a stochastic block model on 256 nodes (all parameters can be seen in title of figure). In order to generate smooth signals X we low-pass filter random Gaussian noise on the graph with a Tikhonov filter.

The resulting learned weight matrix W, after thresholding, is show in the figure of the 5 methods and the ground truth.



Visualization of the learned weight matrix from smooth signals on the SBM

One observation is obvious, which is that the block diagonal structure of the matrix was recovered by each method, regardless of the other patterns. Indeed, if you look closely it appears that the image with the most irregular noise is the ground truth, which was not reconstructed by any of the other methods. There is a clear off diagonal pattern that exists in the Gaussian RBF, and some of it bleeds over into the adaptive kernel. If I had to choose based on my eyes alone, I would say GSP 2, and GSP 1 looks as if its just an attenuated version that could be re-scaled. However, we don't have to choose on eyes alone, that's why we have metrics. To summarize the fit we report the F1 score, which is an intelligent mix of precision and recall. In statistics, recall is the ability of a model to find all the relevant cases in a data-set, whereas precision is the mark of only identifying the relevant data points.

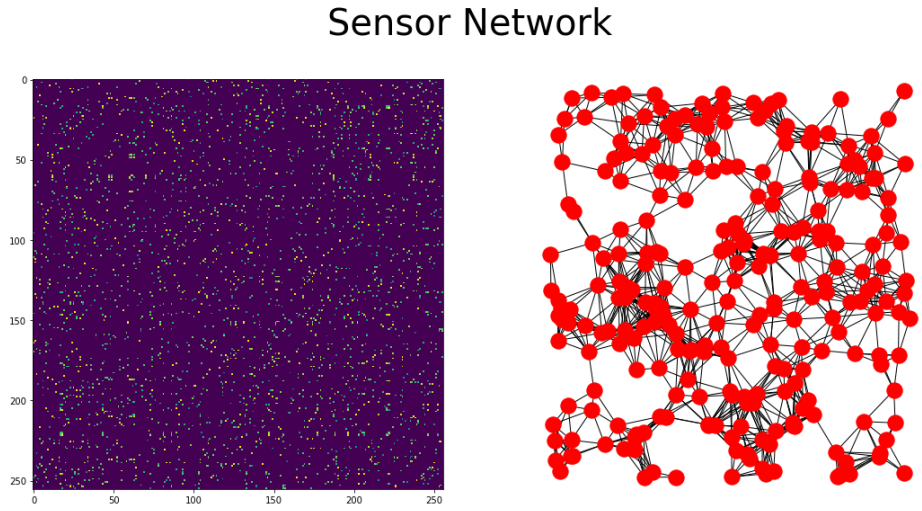
Remember, larger scores are good for F scores. And here we have the simplest K-nearest-neighbors outperforming the algorithm we spent all paper setting up! This isn't good, and I don't have a quick answer for this, besides the obvious I did something wrong in my code or there is some scaling/threshold error going on. At the very least we can say K-n-n make sense for something that might split up into clusters, as you would imagine a lot of mutual nearest neighbor occurrences, and unlikely for there to be matches outside a cluster.

Table 1: F Scores

Model	Score
GSP1	0.10
GSP2	0.08
KNN	0.43
Adaptive	0.06
Gaussian	0.07

4.1.2 Sensor Network

Next we have a network with a very different structure from the SBM, one that appears very difficult to categorize from a glance.



This graph is also on 256 nodes and it is implemented in PyGSP. Achieving the smooth signals the same way as before, we get to our visualization plot like:

which tells us a lot less structure than the last plot I'd have to say. The F scores are similar, like:

Except, of course, the KNN scored even higher. While I was real proud to even get the algorithm implemented and running in python, needless to say the results are a little less than stellar.

4.2 Light at the end of the tunnel?

The one redeeming result comes from some visualization in the networkX package. Recall that we could view the laplacian smoothing as a weighted sparsity constraint. Well, in the

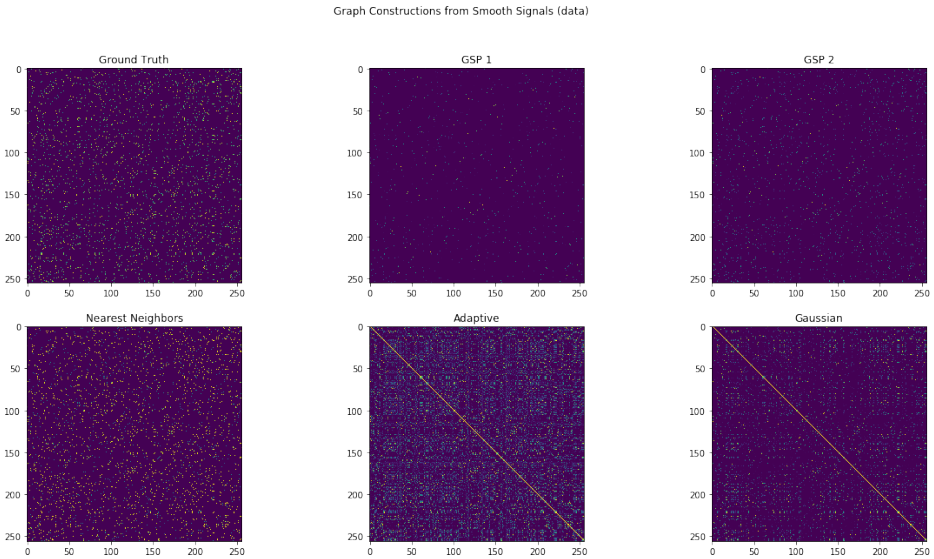
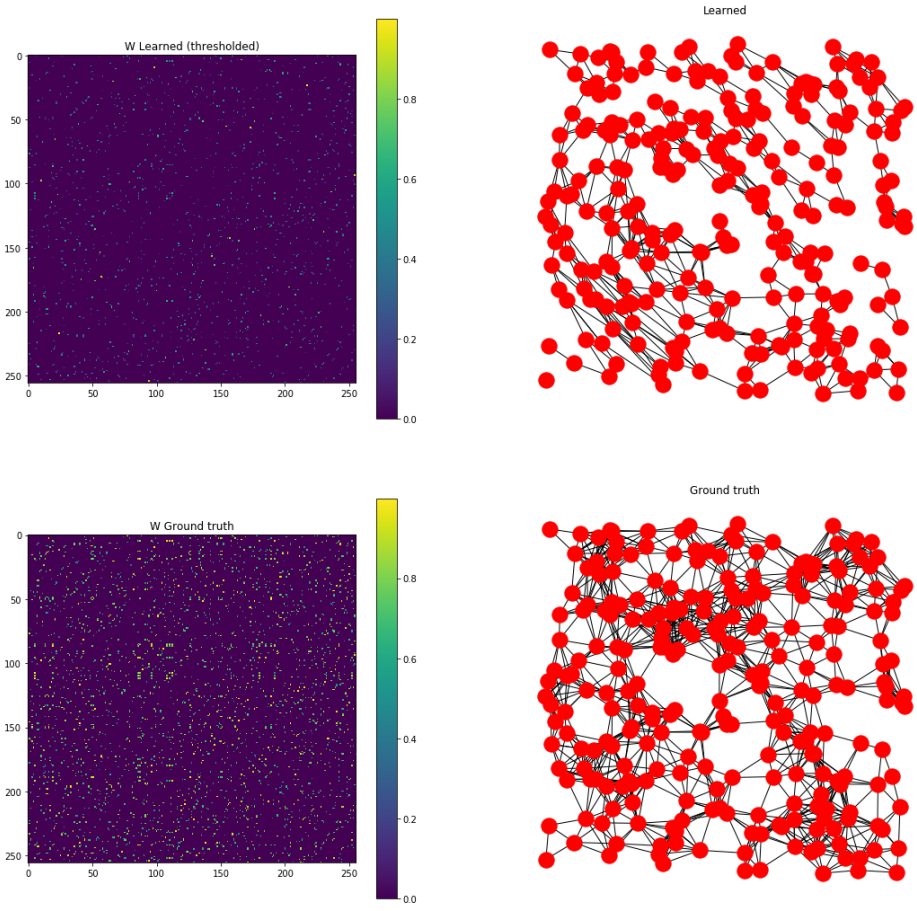


Table 2: F Scores

Model	Score
GSP1	0.10
GSP2	0.08
KNN	0.76
Adaptive	0.07
Gaussian	0.05

visualization between GSP2 and the ground truth graph, we see what might be akin to a sparsification of the original:



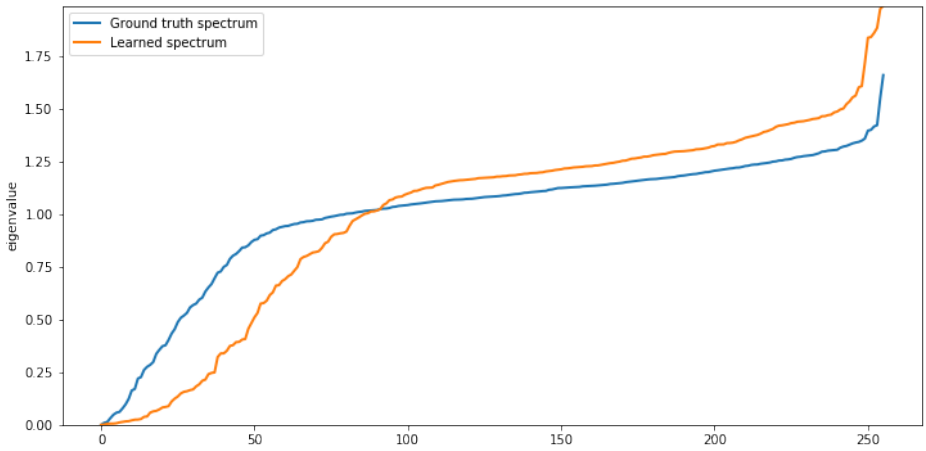
Also, we can look at the eigenvalue spectrum of the learned vs. original:

5 Conclusion

5.1 Big picture overview

Improvement? In terms of time complexity, it takes much longer on a standard laptop to solve for these GSP optimization graphs than it does to construct k-nearest-neighbors or an RBF kernel graph. So we must get something in return for the price of performance. As of now, I'm not sure I see the clear application area where it would be worth it – of course a graph chosen better sounds better, but I am much more pessimistic now than when I started this project.

Hyperparameter Search Of course, I'll readily admit it could be my inability to choose the right hyper-parameters or a misunderstanding of scale effects that led to the poor showing.



Given more time, I think I could figure it out, but of course this is a finite time project.

Scalability To use a cliche thrown around too much in tech these days, "does it scale"? I'm afraid the answer is unclear, but as of now it's not looking great. While tried on relatively modest datasets above, such that one could easily run it on a laptop to evaluate the methods without getting lost in huge datasets, the computation took longer than I had been expecting.

5.2 Future work ideas

Combo Problems Going back and forth between learning the graph and smoothing the signal. Like in Dong for example.

Software Packages In terms of tool building, some of these graphs (or other similar graph learning papers that have come out very recently) might be good candidates to add to graph building packages, like pyGSP or the lab's own graph-tools. Since building a good graph is an input to many other interesting algorithms, having a well-implemented version could be influential for other projects. Some may say this is a less exciting result, as these graph learning methods maybe don't produce interesting output on their own but are useful as something to be fed into future methods; on the contrary, that is why I find them even more exciting, and why, if any of these methods really are better than the state of the art, they should be included to improve quality in data pipelines.

Approximation and optimization If, as suggested above, these methods did become part of software packages, it seems obvious that code would need to be cleaned up and probably optimized, in your language of choice. Then, if there are any real mathematical or physical constraints on the performance of these algorithms beyond test data, perhaps approximation methods would need to be performed. I'm thinking in the lines of fast Fourier transform, truncated SVD, compressive clustering, etc. Especially since these methods would appear before others in some data pipeline, and the current methods of Gaussian RBF are

running fast enough for whatever use case, it would need to be shown that the performance cost really isn't so great compared to increased quality. A good comparison maybe the kmeans++ initialization, which is of course more costly than random initialization but leads to improvements in quality.

6 Biggest Challenges in Project

Optimization and ConvOpt I don't have much optimization experience, so just trying to grok all the optimization going on for graph learning was difficult. The setup of an objective function and some constraints are pretty easy for me to understand when given context, but the method of solving the problem and the convex programming techniques were unfamiliar to me. I'm happy to have learned about them, but I did have to rely heavily on tutorials and code snippets about/from the various convex optimization packages in Python. Of special interest is cvxpy, which has a very appealing model-declarative syntax. However, it is less mature and I could find less written in/about it for help, so I wne t with things I could find more relevant to the problem so the "research" aspect of the project trumped any sort of "code cleverness", at the risk of getting lost in it.

Parameter and Hyper-parameter Tuning I don't think I'm the only one who finds this difficult, and sometimes more of an art than a science.

7 References

[1] High-Order Regularization on Graphs. Denny Zhou, Christopher J.C. Burges. MLG-2008: 6th International Workshop on Mining and Learning with Graphs, Helsinki, Finland. July 2008

[2] Dong, Xiaowen, et al. "Learning Laplacian matrix in smooth graph signal representations." IEEE Transactions on Signal Processing 64.23 (2016): 6160-6173.

[3] Kalofolias, Vassilis (2016). "How to learn a graph from smooth signals". In: Artificial Intelligence and Statistics, pp. 920–929.

[4] Perraudin, Nathanael et al. (2014). "GSPBOX: A toolbox for signal processing on graphs". In: ArXiv e-prints. arXiv: 1408.5781 [cs.IT].

[5] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. IEEE Signal Processing Magazine, 30(3):83–98, 2013.

[6] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In Advances in neural information processing systems, pages 849–856, 2002.

[7] Combettes, Patrick L and Jean-Christophe Pesquet (2011). "Proximal splitting methods in signal processing". In: Fixed-point algorithms for inverse problems in science and engineering. Springer, pp. 185–212.

[8] Daitch, Samuel I, Jonathan A Kelner, and Daniel A Spielman (2009). "Fitting a graph to vector data". In: Proceedings of the 26th Annual International Conference on Machine Learning. ACM, pp. 201–208.