What is the "right" graph?: Simultaneous Graph Structure and Semi-Supervised Learning

Xuan Wu* Lingxiao Zhao*
School of Computer Science
Carnegie Mellon University
{xuanw1,lingxia1}@andrew.cmu.edu

Leman Akoglu
H. John Heinz III College
Carnegie Mellon University
lakoglu@andrew.cmu.edu

ABSTRACT

Semi-supervised learning (SSL) is effectively used for numerous classification problems, thanks to its ability to make use of abundant unlabeled data. The main assumption of various SSL algorithms is that the nearby points on the data manifold likely share a label. Graph-based SSL constructs a graph from point-cloud data as an approximation to the underlying manifold, followed by label inference. It is no surprise that the quality of the constructed graph in capturing the essential structure of the data is critical to the accuracy of the subsequent inference step [6].

How should one construct a graph from the input point-cloud data for graph-based SSL? In this work we introduce a new, parallel graph learning framework (called PG-learn) for the graph construction step of SSL. Our solution has two main ingredients: (1) a gradient-based optimization of the edge weights (more specifically, different kernel bandwidths in each dimension) based on a validation loss function, and (2) a parallel hyperparameter search algorithm with an adaptive resource allocation scheme. In essence, (1) allows us to search around a (random) initial hyperparameter configuration for a better one with lower validation loss. Since the search space of hyperparameters is huge for high-dimensional problems, (2) empowers our gradient-based search to go through as many different initial configurations as possible, where runs for relatively unpromising starting configurations are terminated early to allocate the time for others. As such, PG-learn is a carefully-designed hybrid of random and adaptive search. Through experiments on multi-class classification problems, we show that PG-learn is significantly superior to a list of existing graph construction schemes in accuracy per fixed time budget for hyperparameter tuning, and scales more effectively to high dimensional problems.

1 INTRODUCTION

Graph-based semi-supervised learning (SSL) algorithms, based on graph min-cuts [2], local and global consistency [20], and harmonic energy minimization [21], have been used widely for classification and regression problems. These employ the manifold assumption to take advantage of the unlabeled data, which dictates the label (or value) function to change smoothly on the data manifold.

The data manifold is modeled by a graph structure. In some cases, this graph is explicit; for example, explicit social network connections between individuals have been used in predicting their political orientation [4], age [16], income [8], occupation [17], etc. In others (e.g., image classification), the data is in (feature) vector form, where a graph is to be constructed from point-cloud data. In this graph, nodes correspond to labeled and unlabeled data points and edge weights encode pairwise similarities. Often, some graph sparsification scheme is also used to ensure that the SSL algorithm runs efficiently. Then, labeling is done in such a way that instances connected by large weights are assigned similar labels.

In essence, graph-based SSL for non-graph data consists of two steps: (1) graph construction, and (2) label inference. It is well-understood in areas such as clustering and outlier detection that the choice of the similarity measure has considerable effect on the outcomes. Concretely, Maier et al. demonstrate the critical influence of graph construction on graph-based clustering [15]. Graph-based SSL is no exception. A similar study by de Sousa et al. find that "SSL algorithms are strongly affected by the graph sparsification parameter value and the choice of the adjacency graph construction and weighted matrix generation methods" [6].

Interestingly, however, the (1)st step—graph construction for SSL—is notably under-emphasized in the literature as compared to the (2)nd step—label inference algorithms. Most practitioners default to using a similarity measure such as radial basis function (RBF), coupled with sparsification by ϵ-neighborhood (where node pairs only within distance ϵ are connected) or kNN (where each node is connected to its k nearest neighbors). Hyper-parameters, such as RBF bandwidth σ and ϵ (or k), are then selected by grid search based on cross-validation error.

There exist some work on graph construction beyond ϵ- and kNN-graphs. The b-matching method [11] creates a balanced graph where each node has the same degree. Regression-based approaches [3, 5, 18] assume each instance to be a weighted linear combination of other data points and connect those with non-zero coefficients. Those are methods with strong assumptions about the graph structure. The original work on SSL based on harmonic energy minimization [21] learns different RBF bandwidths $\sigma_{1:d}$ per dimension by minimizing the entropy on unlabeled instances. More recently, adaptive edge weighting [12] is proposed to estimate $\sigma_{1:d}$'s through a local linear reconstruction error minimization. All of these are unsupervised techniques that do not leverage the available labeled data for learning the graph. On the supervised side, labeled data has been used for learning the distance metric [7], which however does not scale to very high dimensional problems. LOOHL [19], a supervised follow up to [21], targeted learning σ_d's via minimizing the leave-one-out prediction error on labeled instances. Despite the non-convexity of their objectives and the large hyperparameter space in high dimensions, all of [12, 19, 21] report results based on a handful of random initializations. In such settings, however, the search space should be explored more strategically—which is exactly what we address in this work.

* First two authors contributed equally to this work.

KDD 2018, London, UK
2018. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Concretely, we address the problem of graph (structure) learning for SSL, suitable and scalable to high dimensional problems. We set out to perform the graph learning and inference steps of semi-supervised learning *simultaneously*. To this end, we learn different RBF bandwidths $\sigma_{1:d}$ for each dimension, by adaptively minimizing a function of validation loss using (iterative) gradient descent. In essence, these different bandwidths become model hyperparameters that provide a more general edge weighting function, which in turn can more flexibly capture the underlying data manifold. Moreover, it is a form of feature selection/importance learning that becomes essential in high dimensions with noisy features. On the other hand, this introduces a scale problem for high-dimensional datasets as we discussed earlier, that is, a large search space with numerous hyperparameters to tune.

Our solution to the scale problem is a Parallel Graph Learning algorithm, called PG-LEARN, which is a hybrid of random search and adaptive search. It is motivated by the successive halving strategy [13], which has been recently proposed for efficient hyperparameter optimization for *iterative* machine learning algorithms. The idea is to start with N hyperparameter settings in parallel threads, adaptively update them for a period of time (in our case, via gradient iterations), discard the worst $N/2$ (or some other fraction) based on validation error, and repeat this scheme for a number of rounds until the time budget is exhausted. In our work, we utilize the idle threads whose hyperparameter settings have been discarded by starting new random configurations on them. Using this scheme, our search tries various random initializations but instead of adaptively updating them fully to completion (i.e., gradient descent convergence), it early-quits those whose progress is not promising (relative to others). While promising configurations are allocated more time for adaptive updates, the time saved from those early-terminations are utilized to initiate new initializations, empowering us to efficiently navigate the search space.

Our main contributions are summarized as follows.

- Motivated by studies that show the critical impact of graph construction on clustering and classification problems [6, 15], we propose a new solution for learning the graph structure from point-cloud data for graph-based SSL, which is suitable and scalable to high dimensional problems.
- We propose to learn a different kernel bandwidth per dimension as a feature importance weighting mechanism to flexibly capture the data manifold, and a gradient based learning of $\sigma_{1:d}$'s that adaptively minimizes a validation loss function.
- In high dimensions, it becomes critical to effectively explore the (large) search space. To this end, we couple our (1) *iterative/sequential*, gradient based, *local* search with (2) a *parallel, resource-adaptive, random* search scheme. In this hybrid, the gradient search runs in parallel with different random initializations, the relatively unpromising fraction of which are terminated early to allocate the time for other random initializations in the search space. In effect, (2) empowers (1) to explore the search space more efficiently.
- We show that our overall solution, called PG-LEARN (for Parallel Graph Learning) outperforms competing graph construction schemes significantly in terms of test accuracy per fixed time budget for hyperparameter search, and further tackles high dimensional, noisy problems more effectively.

2 PRELIMINARIES AND BACKGROUND

2.1 Notation

Consider $\mathcal{D} := \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_l, y_l), \boldsymbol{x}_{l+1}, \ldots, \boldsymbol{x}_{l+u}\}$, a data sample in which the first l examples are labeled, i.e., $\boldsymbol{x}_i \in \mathbb{R}^d$ has label $y_i \in \mathbb{N}_c$ where c is the number of classes and $\mathbb{N}_c := \{p \in \mathbb{N}^* | 1 \leq p \leq c\}$. Let $u := n - l$ be the number of unlabeled examples and $\boldsymbol{Y} \in \mathbb{B}^{n \times c}$ be a binary label matrix in which $\boldsymbol{Y}_{ij} = 1$ if and only if $\boldsymbol{x}_i$ has label $y_i = j$.

The semi-supervised learning task is to assign labels $\{y_{l+1} \ldots, y_{l+u}\}$ to the unlabeled instances.

2.2 Graph Construction

A preliminary step to graph-based semi-supervised learning is the construction of a graph from the point-cloud data. The graph construction process generates a graph $\mathcal{G}$ from $\mathcal{D}$ in which each $\boldsymbol{x}_i$ is a node of $\mathcal{G}$. To generate a weighted matrix $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ from $\mathcal{G}$, one uses a similarity function $\mathcal{K} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ to compute the weights $\boldsymbol{W}_{ij} = \mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j)$.

A widely used similarity function is the RBF (or Gaussian) kernel, $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp(-\mathcal{K}^2(\boldsymbol{x}_i, \boldsymbol{x}_j)/(2\sigma^2))$, in which $\sigma \in \mathbb{R}_+^*$ is the kernel bandwidth parameter.

To sparsify the graph, two techniques are used most often. In ϵ-neighborhood (ϵN) graphs, there exists an undirected edge between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ if and only if $\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq \epsilon$, where $\epsilon \in \mathbb{R}_+^*$ is a free parameter. ϵ thresholding is prone to generating disconnected or almost-complete graphs for an improper value of ϵ. On the other hand, in the k nearest neighbors (kNN) approach, there exists an undirected edge between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ if either $\boldsymbol{x}_i$ or $\boldsymbol{x}_j$ is one of the k closest examples to the other. kNN approach has the advantage of being robust to choosing an inappropriate fixed threshold.

In this work, we use a general kernel function that enables a more flexible graph family, in particular

$$\mathcal{K}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(- \sum_{m=1}^{d} \frac{(\boldsymbol{x}_{im} - \boldsymbol{x}_{jm})^2}{\sigma_m^2} \right), \tag{1}$$

where $\boldsymbol{x}_{im}$ is the m^{th} component of $\boldsymbol{x}_i$. We denote $\boldsymbol{W}_{ij} = \exp\big(- (\boldsymbol{x}_i - \boldsymbol{x}_j)^T \boldsymbol{A} \, (\boldsymbol{x}_i - \boldsymbol{x}_j) \big)$, where $\boldsymbol{A} := diag(\boldsymbol{a})$ is a diagonal matrix with $\boldsymbol{A}_{mm} = a_m = 1/\sigma_m^2$, that corresponds to a metric in which different dimensions/features are given different "weights", which allows a form of feature selection.[1] In addition, we employ kNN graph construction for sparsity.

Our goal is to *learn* both k as well as all the a_m's, by means of which we aim to construct a graph that is suitable for the semi-supervised learning task at hand.

2.3 Graph based Semi-Supervised Learning

Given the constructed graph $\mathcal{G}$, a graph-based SSL algorithm uses $\boldsymbol{W}$ and the label matrix $\boldsymbol{Y}$ to generate output matrix $\boldsymbol{F}$ by label diffusion in the weighted graph. Note that this paper focuses on the multi-class classification problem, hence $\boldsymbol{F} \in \mathbb{R}^{n \times c}$.

There exist a number of SSL algorithms with various objectives. Perhaps the most widely used ones include the Gaussian Random

[1] Setting $\boldsymbol{A}$ equal to (i) the identity, (ii) the (diagonal) variance, or (iii) the covariance matrix would compute similarity based on Euclidean, normalized Euclidean, or Mahalanobis distance, respectively.

Fields algorithm by Zhu et al. [21], Laplacian Support Vector Machine algorithm by Belkin et al. [1], Robust Multi-class Graph Transduction algorithm by Liu and Chang [14], and Local and Global Consistency (LGC) algorithm by Zhou et al. [20].

The topic of this paper is how to effectively learn the hyperparameters of graph construction. Therefore, we focus on how the performance of a given recognized SSL algorithm can be improved by means of learning the graph, rather than comparing the performance of different semi-supervised or supervised learning algorithms. To this end, we use the LGC algorithm which we briefly review here. It is possible to adapt the graph learning ideas introduced in this paper to the context of other SSL algorithms which we do not pursue further.

The LGC algorithm [20] solves the optimization problem

$$\arg\min_{\boldsymbol{F}\in\mathbb{R}^{n\times c}} tr((\boldsymbol{F}-\boldsymbol{Y})^T(\boldsymbol{F}-\boldsymbol{Y}) + \alpha \boldsymbol{F}^T\boldsymbol{L}\boldsymbol{F}) \ , \tag{2}$$

where $tr()$ denotes matrix trace, $\boldsymbol{L} := \boldsymbol{I}_n - \boldsymbol{P}$ is the normalized graph Laplacian, such that $\boldsymbol{I}_n$ is the n-by-n identity matrix, $\boldsymbol{P} = \boldsymbol{D}^{-1/2}\boldsymbol{W}\boldsymbol{D}^{-1/2}$, $\boldsymbol{D} := diag(\boldsymbol{W}\boldsymbol{1}_n)$ and $\boldsymbol{1}_n$ is the n-dimensional all-1's vector. Taking the derivative w.r.t. $\boldsymbol{F}$ and reorganizing the terms, we would get the closed-form solution $\boldsymbol{F} = (\boldsymbol{I}_n + \alpha\boldsymbol{L})^{-1}\boldsymbol{Y}$.

The solution can also be found without explicitly taking any matrix inverse and instead using the power method [9], as

$$\begin{aligned}(\boldsymbol{I}+\alpha\boldsymbol{L})\boldsymbol{F} = \boldsymbol{Y} \Rightarrow & \boldsymbol{F} + \alpha\boldsymbol{F} = \alpha\boldsymbol{P}\boldsymbol{F} + \boldsymbol{Y} \Rightarrow \boldsymbol{F} = \frac{\alpha}{1+\alpha}\boldsymbol{P}\boldsymbol{F} + \frac{1}{1+\alpha}\boldsymbol{Y} \\ \Rightarrow & \ \boldsymbol{F}^{(t+1)} \leftarrow \mu\boldsymbol{P}\boldsymbol{F}^{(t)} + (1-\mu)\boldsymbol{Y} \ . \end{aligned} \tag{3}$$

2.4 Problem Statement

In this work, we address the problem of graph learning. That is, our goal is to learn the hyperparameters of graph construction. Specifically, we want to infer

- $\boldsymbol{A}$, containing the bandwidths (or weights) a_m's for different dimensions in Eq. (1), as well as
- k, for sparse kNN graph construction;

so as to better align the graph structure with the data and the given SSL task.

3 PROPOSED METHOD: PG-LEARN

In this section, we present the formulation and efficient computation of our graph learning algorithm PG-LEARN, for Parallel Graph Learning for SSL.

In essence, the feature weights a_m's and k are the model parameters that govern how the algorithm's performance generalizes to unlabeled data. Typical model selection approaches include random search or grid search to find a configuration of the hyperparameters that yield the best cross-validation performance.

Unfortunately, the search space becomes prohibitively large for high-dimensional datasets that could render such methods futile. In such cases, one could instead carefully select the configurations in an adaptive manner. The general idea is to impose a smooth loss function $g(\cdot)$ on the validation set over which $\boldsymbol{A}$ can be estimated using a gradient based method.

We present the main steps of our algorithm for adaptive hyperparameter search in Algorithm 1.

Algorithm 1 GRADIENT (for Adaptive Hyperparameter Search)

1: Initialize k and $\boldsymbol{a}$ (vector containing a_m's); $t := 0$
2: **repeat**
3: Compute $\boldsymbol{F}^{(t)}$ using kNN graph on current a_m's by (3)
4: Compute gradient $\frac{\partial g}{\partial a_m}$ based on $\boldsymbol{F}^{(t)}$ by (5) for each a_m
5: Update a_m's by $\boldsymbol{a}^{(t+1)} := \boldsymbol{a}^{(t)} - \gamma\frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{a}}$; $t := t+1$
6: **until** a_m's have converged

The initialization in step 1 can be done using some heuristics, although the most prevalent and easiest approach is a random guess. Given a fixed initial (random) configuration, we essentially perform an adaptive search that strives to find a better configuration around it, guided by the validation loss $g(\cdot)$. In Section 3.1, we introduce the specific function $g(\cdot)$ that we use and how to compute its gradient.

While the gradient based optimization is likely to find a better configuration than where it started, the final performance of the SSL algorithm depends considerably on the initialization. Provided that the search space is quite large for high dimensional datasets, it is of paramount importance to try different random initializations in step 1, in other words, to run Algorithm 1 several times. As such, *the* GRADIENT *algorithm can be seen as an adaptive local search*, where we start at a random configuration and adaptively search in the vicinity for a better one.

As we discuss in Section 3.1, the gradient based updates are computationally demanding. This makes naïvely running Algorithm 1 several times expensive. There are however two properties that we can take considerable advantage of: (1) both the SSL algorithm (using the power method) as well as the gradient optimization are *iterative*, *any-time* algorithms (i.e., they can return an answer at any time that they are probed), and (2) different initializations can be run independently in *parallel*.

In particular, our search strategy is inspired by a general framework of parallel hyperparameter search designed for *iterative* machine learning algorithms that has been recently proposed by Jamieson and Talwalkar [10] and a follow-up by Li et al. [13]. This framework perfectly suits our SSL setting for the reasons (1) and (2) above. The idea is to start multiple (random) configurations in parallel threads, run them for a bounded amount of time, probe for their solutions, throw out the worst half (or some other prespecified fraction), and repeat until one configurations remains. By this strategy of early termination, that is by quitting poor initializations early without running them to completion, the compute resources are effectively allocated to promising hyperparameter configurations. Beyond what has been proposed in [10], we start new initializations on the idle threads whose jobs have been terminated in order to fully utilize the parallel threads. We describe the details of our parallel search in Section 3.2.

3.1 Validation Loss $g(\cdot)$ & Gradient Updates

We base the learning of the hyperparameters of our kernel function (a_m's in Eq. (1)) on minimizing some loss criterion on validation data. Let $\mathcal{L} \subset \mathcal{D}$ denote the set of l labeled examples, and $\mathcal{V} \subset \mathcal{L}$ a subset of the labeled examples designated as validation samples. A simple choice for the validation loss would be the labeling error, written as $g_{\boldsymbol{A}}(\mathcal{V}) = \sum_{v\in\mathcal{V}}(1 - \boldsymbol{F}_{vc_v})$, where c_v denotes the true class index for a validation instance v. Other possible choices for each v include $-\log \boldsymbol{F}_{vc_v}$, $(1-\boldsymbol{F}_{vc_v})^x$, $x^{-\boldsymbol{F}_{vc_v}}$, with $x > 1$.

In semi-supervised learning the labeled set is often small. This means the number of validation examples is also limited. To squeeze the most out of the validation set, we propose to use a *pairwise* learning-to-rank objective:

$$g_{\boldsymbol{A}}(\mathcal{V}) = \sum_{c'=1}^{c} \sum_{\substack{(v,v'):\, v\in\mathcal{V}_{c'},\\ v'\in\mathcal{V}\setminus\mathcal{V}_{c'}}} -\log\sigma(\boldsymbol{F}_{vc'} - \boldsymbol{F}_{v'c'}) \tag{4}$$

where $\mathcal{V}_{c'}$ denotes the validation nodes whose true class index is c' and $\sigma(x) = \frac{\exp(x)}{1+\exp(x)}$ is the sigmoid function. The larger the difference $(\boldsymbol{F}_{vc'} - \boldsymbol{F}_{v'c'})$, or intuitively the more *confidently* the solution $\boldsymbol{F}$ ranks validation examples of class c' above other validation examples not in class c', the better it is; since then $\sigma(\cdot)$ would approach 1 and the loss to zero.

In short, we aim to find the hyperparameters $\boldsymbol{A}$ that minimize the total negative log likelihood of ordered validation pairs. The optimization is conducted by gradient descent. The gradient is computed as

$$\begin{aligned}\frac{\partial g}{\partial a_m} &= \frac{\partial\Big(\sum_{(v,v'):v\in\mathcal{V}_{c'},v'\in\mathcal{V}\setminus\mathcal{V}_{c'}} -\boldsymbol{F}_{vv'} + \log(1+\exp(\boldsymbol{F}_{vv'}))\Big)}{\partial a_m} \\ &= \sum_{(v,v'):v\in\mathcal{V}_{c'},v'\in\mathcal{V}\setminus\mathcal{V}_{c'}} (o_{vv'}-1)\Big(\frac{\partial \boldsymbol{F}_{vc'}}{\partial a_m} - \frac{\partial \boldsymbol{F}_{v'c'}}{\partial a_m}\Big)\end{aligned} \tag{5}$$

where we denote by $\boldsymbol{F}_{vv'} = (\boldsymbol{F}_{vc'} - \boldsymbol{F}_{v'c'})$ and $o_{vv'} = \sigma(\boldsymbol{F}_{vv'})$.

The values $\frac{\partial \boldsymbol{F}_{vc'}}{\partial a_m}$ and $\frac{\partial \boldsymbol{F}_{v'c'}}{\partial a_m}$ for each class c' and $v, v' \in \mathcal{V}$ can be read off of matrix $\frac{\partial \boldsymbol{F}}{\partial a_m}$, which is given as

$$\frac{\partial \boldsymbol{F}}{\partial a_m} = -(\boldsymbol{I}_n + \alpha\boldsymbol{L})^{-1}\frac{\partial(\boldsymbol{I}_n+\alpha\boldsymbol{L})}{\partial a_m}\boldsymbol{F} = \alpha(\boldsymbol{I}_n+\alpha\boldsymbol{L})^{-1}\frac{\partial \boldsymbol{P}}{\partial a_m}\boldsymbol{F}\,, \tag{6}$$

using the equivalence $d\boldsymbol{X}^{-1} = -\boldsymbol{X}^{-1}(d\boldsymbol{X})\boldsymbol{X}^{-1}$. Recall that $\boldsymbol{P} = \boldsymbol{D}^{-1/2}\boldsymbol{W}\boldsymbol{D}^{-1/2}$ with $\boldsymbol{P}_{ij} = \frac{\boldsymbol{W}_{ij}}{\sqrt{d_i d_j}}$; d_i being node i's degree in $\mathcal{G}$.

We can then write

$$\frac{\partial \boldsymbol{P}_{ij}}{\partial a_m} = \frac{\partial \boldsymbol{W}_{ij}}{\partial a_m}\frac{1}{\sqrt{d_i d_j}} - \frac{\boldsymbol{W}_{ij}}{2}(d_i d_j)^{-3/2}\frac{\partial d_i d_j}{\partial a_m} \tag{7}$$

$$= \frac{\partial \boldsymbol{W}_{ij}}{\partial a_m}\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}} - \frac{\boldsymbol{W}_{ij}}{2}\Big(\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}}\Big)^3\Big(d_j\frac{\partial d_i}{\partial a_m} + d_i\frac{\partial d_j}{\partial a_m}\Big) \tag{8}$$

$$= \frac{\partial \boldsymbol{W}_{ij}}{\partial a_m}\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}} - \frac{\boldsymbol{W}_{ij}}{2}\Big(\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}}\Big)^3 \Big(\sum_n \boldsymbol{W}_{in}\cdot\sum_n\frac{\partial \boldsymbol{W}_{jn}}{\partial a_m} + \sum_n \boldsymbol{W}_{jn}\cdot\sum_n\frac{\partial \boldsymbol{W}_{in}}{\partial a_m}\Big) \tag{9}$$

3.1.1 Matrix-form gradient. We can rewrite all element-wise gradients into a combined matrix-form gradient. The matrix-form is compact and can be computed more efficiently on platforms optimized for matrix operations (e.g., Matlab).

The matrix-form uses 3-d matrix (or tensor) representation. In the following, we use $\odot$ to denote element-wise multiplication, $\oslash$ element-wise division, and $\otimes$ for element-wise power. In addition, $\cdot$ denotes regular matrix dot product. For multiplication and division, a 3-d matrix should be viewed as a 2-d matrix with vector elements.

First we extend the derivative w.r.t. a_m in Eq. (9) into derivative w.r.t. $\boldsymbol{a}$:

$$\frac{\partial \boldsymbol{P}_{ij}}{\partial \boldsymbol{a}} = \frac{\partial \boldsymbol{W}_{ij}}{\partial \boldsymbol{a}}\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}} - \frac{\boldsymbol{W}_{ij}}{2}\Big(\frac{\boldsymbol{P}_{ij}}{\boldsymbol{W}_{ij}}\Big)^3 \Big(\sum_n \boldsymbol{W}_{in}\cdot\sum_n\frac{\partial \boldsymbol{W}_{jn}}{\partial \boldsymbol{a}} + \sum_n \boldsymbol{W}_{jn}\cdot\sum_n\frac{\partial \boldsymbol{W}_{in}}{\partial \boldsymbol{a}}\Big) \tag{10}$$

To write this equation concisely, let 3d-matrix Ω be $\frac{\partial \boldsymbol{W}}{\partial \boldsymbol{a}}$ with vector elements $\Omega_{ij} = \frac{\partial \boldsymbol{W}_{ij}}{\partial \boldsymbol{a}}$, and let 3d-matrix $\Delta\boldsymbol{X}$ be the one with vector elements $\Delta\boldsymbol{X}_{ij} = (\boldsymbol{x}_i - \boldsymbol{x}_j)^2$.

Then we can rewrite some equations using the above notation:

$$\sum_n \boldsymbol{W}_{in} = (\boldsymbol{W}\cdot\boldsymbol{1}_n)_i \tag{11}$$

$$\sum_n \frac{\partial \boldsymbol{W}_{jn}}{\partial \boldsymbol{a}} = (\Omega\cdot\boldsymbol{1}_n)_j \tag{12}$$

$$\sum_n \boldsymbol{W}_{in}\cdot\sum_n\frac{\partial \boldsymbol{W}_{jn}}{\partial \boldsymbol{a}} = (\boldsymbol{W}\cdot\boldsymbol{1}_n\cdot(\Omega\cdot\boldsymbol{1}_n)^T)_{ij} \tag{13}$$

Now we can rewrite element-wise gradients in (10) into one matrix-form gradient:

$$\begin{aligned}\frac{\mathrm{d}\boldsymbol{P}}{\mathrm{d}\boldsymbol{a}} &= \Omega\odot(\boldsymbol{P}\oslash\boldsymbol{W}) - \frac{1}{2}\boldsymbol{P}^{\otimes 3}\oslash\boldsymbol{W}^{\otimes 2} \\ &\quad\odot(\boldsymbol{W}\cdot\boldsymbol{1}_n\cdot(\Omega\cdot\boldsymbol{1}_n)^T + (\boldsymbol{W}\cdot\boldsymbol{1}_n\cdot(\Omega\cdot\boldsymbol{1}_n)^T)^T)\end{aligned} \tag{14}$$

The only thing left is the computation of $\Omega = \frac{\partial \boldsymbol{W}}{\partial \boldsymbol{a}}$. Notice that

$$\begin{aligned}\frac{\partial \boldsymbol{W}_{ij}}{\partial a_m} &= \frac{\partial \exp(-\sum_{m=1}^d a_m(\boldsymbol{x}_{id}-\boldsymbol{x}_{jd})^2)}{\partial a_m} = -\boldsymbol{W}_{ij}(\boldsymbol{x}_{id}-\boldsymbol{x}_{jd})^2 \\ \Longrightarrow \frac{\partial \boldsymbol{W}_{ij}}{\partial \boldsymbol{a}} &= -\boldsymbol{W}_{ij}(\boldsymbol{x}_i-\boldsymbol{x}_j)^2 = -\boldsymbol{W}_{ij}\Delta\boldsymbol{X}_{ij} \\ \Longrightarrow \frac{\mathrm{d}\boldsymbol{W}}{\mathrm{d}\boldsymbol{a}} &= -\boldsymbol{W}\odot\Delta\boldsymbol{X} = \Omega\end{aligned} \tag{15}$$

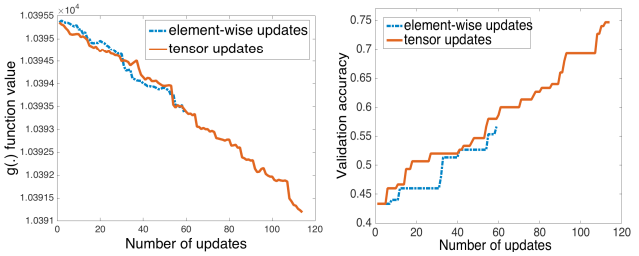
All in all, we transformed the element-wise gradients $\frac{\partial \boldsymbol{P}_{ij}}{\partial a_m}$ as given in Eq. (9) to compact, tensor-form updates $\frac{\mathrm{d}\boldsymbol{P}}{\mathrm{d}\boldsymbol{a}}$ as in Eq. (14).

Empirically, the closed-form updates provided us with two-fold speed up. As an example, Figure 1 (a) shows the decrease in loss $g(\cdot)$ and (b) the increase in validation accuracy over the iterations of Algorithm 1 (avg'ed over 5 random initializations). We see that tensor-form updates can finish 120 iterations as compared to 60 by the element-wise updates within 30 minutes. Also notice that the decrease in our loss function $g(\cdot)$'s value in (a) well-aligns with the increase in validation accuracy in (b).

3.1.2 Computational complexity analysis. Next we give the complexity analysis of computing the gradient $\frac{\mathrm{d}g}{\mathrm{d}\boldsymbol{a}}$ in line 4 of Algorithm 1 as outlined in this subsection, as well as the complexity of constructing the kNN graph and computing $\boldsymbol{F}$ in line 3.

Let us denote the number of non-zeros in $\boldsymbol{W}$, i.e. the number of edges in the kNN graph, by $e = nnz(\boldsymbol{W})$. We assume $kn \le e \le 2kn$ remains near-constant as $\boldsymbol{a}$ changes over the GRADIENT iterations.

To begin, constructing tensor Ω as in Eq. (15) takes $O(ed)$. Computing $\frac{\mathrm{d}\boldsymbol{P}}{\mathrm{d}\boldsymbol{a}}$ as in Eq. (14) is also $O(ed)$. Next, obtaining matrix $\frac{\partial \boldsymbol{F}}{\partial a_m}$ in Eq. (6) seemingly requires inverting $(\boldsymbol{I}_n+\alpha\boldsymbol{L})^{-1}$. However, we



(a) $g_A(F)$ vs. #updates (b) validation acc. vs. #updates

Figure 1: Within the same amount of time (30 mins), we perform twice the number of updates using tensor-form (red) as compared to element-wise (blue) gradients.

can rewrite Eq. (6) as

$$(I_n+\alpha I_n-\alpha P)\frac{\partial F}{\partial a_m} = \alpha\frac{\partial P}{\partial a_m}F \Rightarrow \frac{\partial F}{\partial a_m} = \alpha(P-I_n)\frac{\partial F}{\partial a_m}+\alpha\frac{\partial P}{\partial a_m}F$$

which can be solved via the power method that takes t iterations in $O(ect)$. Computing $\frac{\partial F}{\partial a_m}$ and plugging in Eq. (5) to get $g(\cdot)$'s gradient for all a_m's then takes $O(ectd)$, or equivalently $O(knctd)$.

In line 3, updated a_m's are used to recompute all entries of $\boldsymbol{W}$ in $O(n^2d)$. Then, we find kNNs for each instance in $O(n\log k)$, using a min-heap to maintain the top-k largest similarities, that is $O(n^2\log k)$ for all.[2] Given the kNN graph, $\boldsymbol{F}$ can then be computed via (3) in $O(ect')$ for t' iterations of the power method.

Overall, one iteration of Algo. 1 takes $O(n^2[d+\log k]+knctd)$.

3.2 Parallel Hyperparameter Search with Adaptive Resource Allocation

For high-dimensional datasets, the search space of hyperparameter configurations is huge. In essence, Algorithm 1 is an adaptive search around a single initial point in this space. As with many gradient-based optimization of high-dimensional non-convex functions with unknown smoothness, its performance depends on the initialization. Therefore, trying different initializations of Algorithm 1 is beneficial to improving performance.

An illustrative example over a 2-d search space is shown in Figure 2 (best in color). In this space most configurations yield poor validation accuracy, as would be the likely case in even higher dimensions. In the figure, eight random configurations are shown (with stars). The sequence of arrows from a configuration can be seen analogous to the iterations of a single run of Algorithm 1.

While it is beneficial to try as many random initializations as possible, especially in high dimensions, adaptive search is slow. A single gradient update by Algorithm 1 takes time in $O(ectd)$ followed by reconstruction of the kNN graph. Therefore, it would be good to quit the algorithm early if the ongoing progress is not promising (e.g., poor initializations 6–8 in Figure 2) and simply try a new initialization. This would allow using the time efficiently for going through a larger number of configurations.

One way to realize such a scheme is called successive halving [10], which relies on an early-stopping strategy for iterative machine learning algorithms. The idea is quite simple and follows directly from its name: try out a set of hyperparameter configurations for some fixed amount of time (say in parallel threads), evaluate the performance of all configurations, keep the best half (terminate the worst half of the threads), and repeat until one configuration remains while allocating exponentially increasing amount of time after each round to not-yet-terminated, promising configurations (i.e., threads). Our proposed method is a parallel implementation of their general framework adapted to our problem, and further utilizes the idle threads that have been terminated.

[2] This can be sped up by parallelizing kNN computation for partitions of nodes.

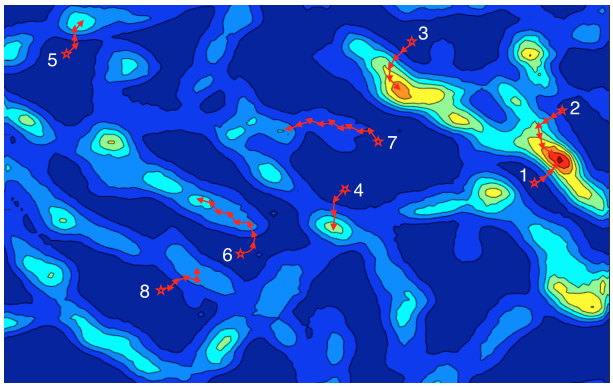


Figure 2: (best in color) The heatmap shows the validation error over an example 2-d search space with red corresponding to areas with lower error. Our approach is an inter-mix of random and adaptive search. We start at various random configurations (stars 1–8) and adaptively improve them (arrows depicting gradient updates), while strategically terminating unpromising ones (like 6, 7, and 8) early.

Algorithm 2 gives the steps of our proposed method PG-LEARN, which calls the GRADIENT subroutine in Algorithm 1. Besides the input dataset $\mathcal{D}$, PG-LEARN requires three inputs: (1) budget B; the maximum number of time units[3] that can be allocated to one thread (i.e., one initial hyperparameter configuration), (2) downsampling rate r; an integer that controls the fraction of threads terminated (or equally, configurations discarded) in each round of PG-LEARN, and finally (3) T; the number of parallel threads.

Concretely, PG-LEARN performs $R = \lfloor \log_r B \rfloor$ rounds of elimination. At each round, the best $1/r$ fraction of configurations are retained. Eliminations are done in exponentially increasing time intervals, that is, first round occurs at time B/r^R, second round at B/r^{R-1}, and so on.

After setting the number of elimination rounds R and the duration of the first round, denoted d_1 (line 1), PG-LEARN starts by obtaining T initial hyperparameter configurations (line 2). Note that a configuration is a $(k, \boldsymbol{a}_{1:d})$ pair. Our implementation[4] of PG-LEARN is parallel. As such, each thread draws their own configuration; uniformly at random. Then, each thread runs the GRADIENT (Algorithm 1) with their corresponding configuration for duration d_1 and returns the validation loss (line 3).

At that point, PG-LEARN enters the rounds of elimination (line 4). L validation loss values across threads are gathered at the master node, which identifies the top $\lfloor T/r \rfloor$ configurations C_{top} (or threads) with the lowest loss (line 5). The master then terminates the runs on the remaining threads and restarts them afresh with new configurations C_{new} (line 6). The second round is to run until

[3] We assume time is represented in units, where a unit is the minimum amount of time one would run the models before comparing against each other.

[4] We release all source code at https://github.com/LingxiaoShawn/PG-Learn

Algorithm 2 PG-LEARN (for Parallel Hyperparameter Search)

Input: Dataset $\mathcal{D}$, budget B time units, downsampling rate r (= 2 by default), number of parallel threads T
Output: Hyperparameter configuration $(k, \boldsymbol{a})$

1: $R = \lfloor \log_r B \rfloor$, $d_1 = Br^{-R}$
2: $C :=$ get_hyperparameter_configuration(T)
3: $L := \{$run_GRADIENT_then_return_val_loss$(c, d_1) : c \in C\}$
4: **for** $i \in \{1, \ldots, R\}$ **do**
5: $\quad C_{top} :=$ get_top$(C, L, \lfloor T/r \rfloor)$
6: $\quad C_{new} :=$ get_hyperparameter_configuration$(T - \lfloor T/r \rfloor)$
7: $\quad d_i = B(r^{-(R-i)} - r^{-(R-i+1)})$
8: $\quad L_{top} := \{$resume_GRADIENT_then_return_val_loss(c, d_i) for $c \in C_{top}\}$
9: $\quad L_{new} := \{$run_GRADIENT_then_return_val_loss(c, d_i) for $c \in C_{new}\}$
10: $\quad C := C_{top} \cup C_{new}, \quad L := L_{top} \cup L_{new}$
11: **end for**
12: **return** $c_{top} :=$ get_top$(C, L, 1)$

B/r^{R-1}, or for $B/r^{R-1} - B/r^R$ in duration. After the ith elimination, in general, we run the threads for duration d_i as given in line 7—notice that exponentially increasing amount of time is provided to "surviving" configurations over time. In particular, the threads with the promising configurations in C_{top} are *resumed* their runs from where they are left off with the GRADIENT iterations (line 8). The remaining threads start with the GRADIENT iterations using their new initialization (line 9). Together, this ensures full utilization of the threads at all times. Eliminations continue for R rounds, following the same procedure of resuming best threads and restarting from the rest of the threads (lines 4–11). After round R, all threads run until time budget B at which point the (single) configuration with the lowest validation error is returned (line 12).

The underlying principle of PG-LEARN exploits the insight that a hyperparameter configuration which is destined to yield good performance ultimately is more likely than not to perform in the top fraction of configurations even after a small number of iterations. In essence, even if the performance of a configuration after a small number of iterations of the GRADIENT (Algorithm 1) may not be representative of its ultimate performance after a large number of iterations in *absolute* terms, its *relative* performance in comparison to the alternatives is roughly maintained. We note that different configurations get to run different amounts of time before being tested against others for the first time (depending on the round they get introduced). This diversity offers some robustness against variable convergence rates of the $g(\cdot)$ function at different random starting points.

Example: In Figure 3 we provide a simple example to illustrate PG-LEARN's execution, using $T = 8$ parallel threads, downsampling rate $r = 2$ (equiv. to halving), and $B = 16$ time units of processing budget. There are $\lfloor \log_2 16 \rfloor = 4$ rounds of elimination at $t = 1, 2, 4, 8$ respectively, with the final selection being made at $t = B$. It starts with 8 different initial configurations (depicted with circles) in parallel threads. At each round, bottom half (=4) of the threads with highest validation loss are terminated with their iterations of Algorithm 1 and restart running Algorithm 1 with a new initialization (depicted with a crossed-circle). Overall, $T + (1 - 1/r)T\lfloor \log_r B \rfloor = 8 + 4\lfloor \log_2 16 \rfloor = 24$ configurations are examined—a larger number as compared to the initial 8, thanks to the early-stopping and adaptive resource allocation strategy.

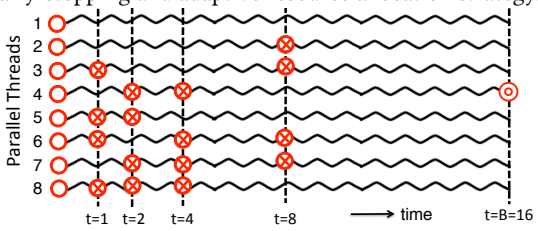


Figure 3: Example execution of PG-LEARN with $T = 8$ parallel threads, downsampling rate $r = 2$, and budget $B = 16$ time units. At each "check point" in time (dashed vertical lines), (worst) half of the runs are discarded and corresponding threads restart Algorithm 1 with new random configurations of $(k, a_{1:d})$. At the end, hyperparameters that yield the lowest $g(\cdot)$ function value (i.e. validation loss) across all threads are returned (those by thread 4 in this example).

Next in Figure 4 we show example runs on two different real-world datasets, depicting the progression of validation (blue) and test (red) accuracy over time, using $T = 32, r = 2, B = 64$; ≈15 sec. unit-time. Thin curves depict those for individual threads. Notice the new initializations starting at different rounds, which progressively improve their validation accuracy over gradient updates (test acc. closely follows). Bold curves depict the overall-best validation accuracy (and corresponding test acc.) across all threads over time.

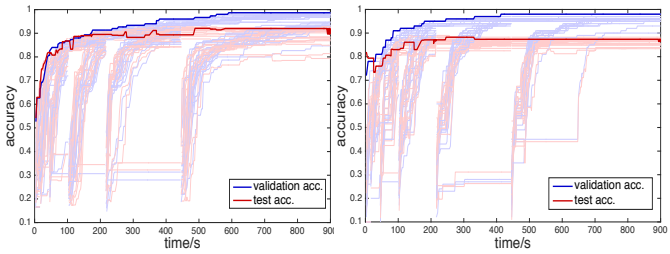


Figure 4: PG-LEARN's val. (blue) and corresponding test (red) acc. vs. time on COIL (left) and MNIST (right) (see Table 1).

Setting T, B, and r: Before we conclude the description of our proposed method, we briefly discuss the choices for its inputs. Number of threads T is a resource-driven input. Depending on the platform being utilized—single machine or a parallel architecture like Hadoop or Spark—PG-LEARN can be executed with as many parallel threads as physically available to the practitioner. Time units B should be chosen based on the upper bound of practically available time. For example, if one has time to run hyperparameter tuning for at most 3 hours and the minimum amount of time that is meaningful to execute gradient search of configurations before comparing them (i.e., unit time) is 5 minutes, then B becomes $180/5 = 36$ units. Finally, r can be seen as a knob for greediness. A larger value of r corresponds to more aggressive elimination with fewer rounds; specifically, each round terminates $T(r-1)/r$ configurations for a total of $\lfloor \log_r B \rfloor$ rounds. All in all, T and B are set based on practical *resource constraints*, physical and temporal, respectively. On the other hand, r can be set to a small integer, like 2 or 3, without results being very sensitive to the choice.

Table 1: Summary of (multi-class) datasets used in this work.

Name	#pts n	#dim d	#cls c	description
COIL	1500	241	6	objects with various shapes
USPS	1000	256	10	handwritten digits
MNIST	1000	784	10	handwritten digits
UMIST	575	644	20	faces (diff. race/gender/etc.)
YALE	320	1024	5	faces (diff. illuminations)

4 EVALUATION

4.1 Datasets and Baselines

Datasets: We use the publicly available multi-class classification datasets listed in Table 1.

COIL[5] (Columbia Object Image Library) contains images of different objects taken from various angles. Features are shuffled and downsampled pixel values from the red channel. USPS[6] is a standard dataset for handwritten digit recognition, with numeric pixel values scanned from the handwritten digits on envelopes from the U.S. Postal Service. MNIST[7] is another popular handwritten digit dataset, containing size-normalized and centered digit images. UMIST[8] face database is made up of images of 20 individuals with mixed race, gender, and appearance. Each individual takes a range of poses, from profile to frontal views. YALE[9] is a subset of the extended Yale Face Database B, which consists of frontal images under different illuminations from 5 individuals.

Baselines: We compare the classification accuracy of PG-LEARN against five baselines:

(1) *Grid* search (GS): k-NN graph with RBF kernel where k and bandwidth σ are chosen via grid search,
(2) *Rand*$_d$ search (RS): k-NN with RBF kernel where k and different bandwidths $\boldsymbol{a}_{1:d}$ are randomly chosen,
(3) *MinEnt*: Minimum Entropy based tuning of $\boldsymbol{a}_{1:d}$'s as proposed by Zhu et al. [21] (generalized to multi-class),
(4) *IDML*: Iterative self-learning scheme combined with distance metric learning by Dhillon et al. [7], and
(5) *AEW*: Adaptive Edge Weighting by Karasuyama et al. [12] that estimates $\boldsymbol{a}_{1:d}$'s through local linear reconstruction.

Note that *Grid* and *Rand*$_d$ are standard techniques employed by practitioners most typically. *MinEnt* is perhaps the first graph-learning strategy for SSL which was proposed as part of the Gaussian Random Fields SSL algorithm. It estimates hyperparameters by minimizing the entropy of the solution on unlabeled instances via gradient updates. *IDML* uses and iteratively enlarges the labeled data (via self-learning) to estimate the metric $\boldsymbol{A}$; which we restrict to a diagonal matrix, as our datasets are high dimensional and metric learning is prohibitively expensive for a full matrix. We generalized these baselines to multi-class and implemented them ourselves. We open-source (1)–(4) along with our PG-LEARN implementation.[4] Finally, *AEW* is one of the most recent techniques on graph learning, which extends the LLE [18] idea by restricting the regression coefficients (i.e., edge weights) to be derived from Gaussian kernels. We use their publicly-available implementation.[10]

[5] http://olivier.chapelle.cc/ssl-book/index.html, see 'benchmark datasets'
[6] http://www.cs.huji.ac.il/~shais/datasets/ClassificationDatasets.html
[7] http://yann.lecun.com/exdb/mnist/
[8] https://www.sheffield.ac.uk/eee/research/iel/research/face
[9] http://www.cad.zju.edu.cn/home/dengcai/Data/FaceData.html
[10] http://www.bic.kyoto-u.ac.jp/pathway/krsym/software/MSALP/MSALP.zip

Table 2: Test accuracy with 10% labeled data, avg'ed across 10 random samples; 15 mins of hyperparameter tuning on single thread. Symbols ▲ (p<0.005) and △ (p<0.01) denote the cases where PG-LEARN is significantly better than the baseline w.r.t. the paired Wilcoxon signed rank test.

Dataset	PG-LRN	*MinEnt*	*IDML*	*AEW*	*Grid*	*Rand*$_d$
COIL	**0.9232**	0.9116▲	0.7508▲	0.9100▲	0.8929▲	0.8764▲
USPS	**0.9066**	**0.9088**	0.8565▲	0.8951▲	0.8732▲	0.8169▲
MNIST	**0.8241**	**0.8163**	0.7801△	0.7828▲	0.7550▲	0.7324▲
UMIST	**0.9321**	0.8954▲	0.8973△	0.8975▲	0.8859▲	0.8704▲
YALE	**0.8234**	0.7648△	0.7331▲	0.7386▲	0.6576▲	0.6797▲

4.2 Empirical Results

4.2.1 Single-thread Experiments. We first evaluate the proposed PG-LEARN against the baselines on a fair ground using a single thread, since the baselines do not leverage any parallelism. Single-thread PG-LEARN is simply the GRADIENT as given in Algo. 1.

Setup: For each dataset, we sample 10% of the points at random as the labeled set $\mathcal{L}$, under the constraint that all classes must be present in $\mathcal{L}$ and treat the remaining unlabeled data as the test set. For each dataset, 10 versions with randomly drawn labeled sets are created and the average test accuracy across 10 runs is reported. Each run starts with a different random configuration of hyperparameters. For PG-LEARN, *Grid*, and *Rand*$_d$, we choose (a small) $k \in [5, 20]$. σ for *Grid* and *MinEnt*[11], and a_m's for PG-LEARN, *Rand*$_d$, and *AEW* are chosen from $[0.1\bar{d}, 10\bar{d}]$, where $\bar{d}$ is the mean Euclidean distance across all pairs. Other hyperparameters of the baselines, like ϵ for *MinEnt* and γ and ρ for *IDML*, are chosen as in their respective papers. Graph learning is performed for 15 minutes, around which all gradient-based methods have converged.

Results: Table 2 gives the average test accuracy of the methods on each dataset, avg'ed over 10 runs with random labeled sets. PG-LEARN outperforms its competition significantly, according to the paired Wilcoxon signed rank test on a vast majority of the cases—only on the two handwritten digit recognition tasks there is no significant difference between PG-LEARN and *MinEnt*. Not only PG-LEARN is significantly superior to existing methods, its performance is desirably high in absolute terms. It achieves 93% prediction accuracy on the 20-class UMIST, and 82% on the 2^{10}-dimensional YALE dataset.

Next we investigate how the prediction performance of the competing methods changes by varying labeling percentage. To this end, we repeat the experiments using up to 50% labeled data. As shown in Figure 5, test error tends to drop with increasing amount of labels as expected. PG-LEARN achieves the lowest error in many cases across datasets and labeling ratios. *MinEnt* is the closest competition on USPS and MNIST, which however ranks lower on UMIST and YALE. Similarly, *IDML* is close competition on UMIST and YALE, which however performs poorly on COIL and USPS. In contrast, PG-LEARN *consistently* performs near the top.

We quantify the above more concretely, and provide in Table 3 the test accuracy for each labeling %, averaged across random samples from all datasets, along with significance test results. We also give the average rank per method, as ranked by test error (hence, lower is better).

[11] *MinEnt* initializes $\boldsymbol{a}$ uniformly, i.e., all a_m's are set to the same σ initially [21].

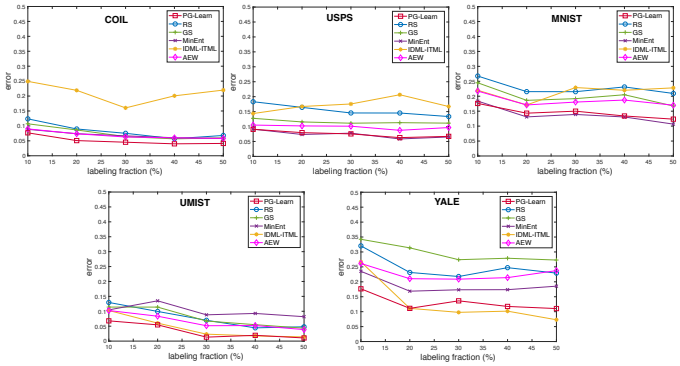


Figure 5: Test error (avg'ed across 3 random samples) as labeled data percentage is increased up to 50%. PG-LEARN performs the best in many cases, and *consistently* ranks in top two among competitors on each dataset and each labeling %.

PG-LEARN significantly outperforms *all* competing methods in accuracy at *all* labeling ratios w.r.t. the paired Wilcoxon signed rank test at $p = 0.01$, as well as achieves the lowest rank w.r.t. test error. On average, *MinEnt* is the closest competition, followed by *AEW*. Despite being supervised, *IDML* does not perform on par. This may be due to labeled data not being sufficient to learn a proper metric in high dimensions, and/or the labels introduced during self-learning being noisy. We also find *Grid* and $Rand_d$ to rank at the bottom, suggesting that learning the graph structure provides advantage over these standard techniques.

Table 3: Average test accuracy and rank (w.r.t. test error) of methods across datasets for varying labeling %. ▲ (p<0.005) and △ (p<0.01) denote the cases where PG-LEARN is significantly better w.r.t. the paired Wilcoxon signed rank test.

Labeled	PG-L	*MinEnt*	*IDML*	*AEW*	*Grid*	$Rand_d$
10% acc.	**0.8819**	0.8594▲	0.8036▲	0.8448▲	0.8129▲	0.7952▲
rank	**1.20**	2.20	4.40	2.80	4.80	5.60
20% acc.	**0.8900**	0.8504▲	0.8118▲	0.8462▲	0.8099▲	0.8088▲
rank	**1.42**	2.83	4.17	2.92	4.83	4.83
30% acc.	**0.9085**	0.8636▲	0.8551▲	0.8613▲	0.8454▲	0.8386▲
rank	**1.33**	3.67	3.83	3.17	4.00	5.00
40% acc.	**0.9153**	0.8617▲	0.8323▲	0.8552▲	0.8381▲	0.8303▲
rank	**1.67**	3.67	3.50	3.67	4.00	4.50
50% acc.	**0.9251**	0.8700△	0.8647▲	0.8635▲	0.8556▲	0.8459▲
rank	**1.50**	3.17	3.83	3.67	4.00	4.83

4.2.2 Parallel Experiments with Noisy Features. Next we fully evaluate PG-LEARN in the parallel setting as proposed in Algo. 2. Graph learning is especially beneficial for SSL in noisy scenarios, where there exist irrelevant or noisy features that would cause simple graph construction methods like kNN and *Grid* go astray. To the effect of making the classification tasks more challenging, we *double* the feature space for each dataset, by adding 100% new noise features with values drawn randomly from standard $Normal(0, 1)$. Moreover, this provides a ground truth on the importance of features, based on which we are able to quantify how well our PG-LEARN recovers the necessary underlying relations by learning the appropriate feature weights.

Setup: We report results comparing PG-LEARN only with *MinEnt*, *Grid*, and $Rand_d$—in this setup, *IDML* failed to learn a metric in several cases due to degeneracy and the authors' implementation[10] of *AEW* gave out-of-memory errors in many cases. This however does not take away much, since *MinEnt* proved to be the second-best after PG-LEARN in the previous section (see Table 3) and *Grid* and $Rand_d$ are the typical methods used often in practice.

Given a budget B units of time and T parallel threads for our PG-LEARN, each competing method is executed for a total of BT units, i.e. all methods receive the same amount of processing time.[12] Specifically, *MinEnt* is started in T threads, each with a random initial configuration that runs until time is up (i.e., to completion, no early-terminations). *Grid* picks (k, σ) from the 2-d grid that we refine recursively, that is, split into finer resolution containing more cells as more allocated time remains, while $Rand_d$ continues picking random combinations of $(k, \boldsymbol{a}_{1:d})$. When the time is over, each method reports the hyperparameters that yield the highest validation accuracy, using which the test accuracy is computed.

Results: Table 4 presents the average test accuracy over 10 random samples from each dataset, using $T = 32$. We find that despite 32× more time, the baselines are crippled by the irrelevant features and increased dimensionality. In contrast, PG-LEARN maintains notably high accuracy that is significantly better than all the baselines on all datasets at $p = 0.01$.

Table 4: Test accuracy on datasets with 100% added noise features, avg'ed across 10 samples; 15 mins of hyperparameter tuning on $T = 32$ threads. Symbols ▲ (p<0.005) and △ (p<0.01) denote the cases where PG-LEARN is significantly better than the baseline w.r.t. the paired Wilcoxon signed rank test.

Dataset	PG-LRN	*MinEnt*	*Grid*	$Rand_d$
COIL	**0.9044**	0.8197▲	0.6311▲	0.6954▲
USPS	**0.9154**	0.8779△	0.8746▲	0.7619▲
MNIST	**0.8634**	0.8006▲	0.7932▲	0.6668▲
UMIST	**0.8789**	0.7756▲	0.7124▲	0.6405▲
YALE	**0.6859**	0.5671▲	0.5925▲	0.5298▲

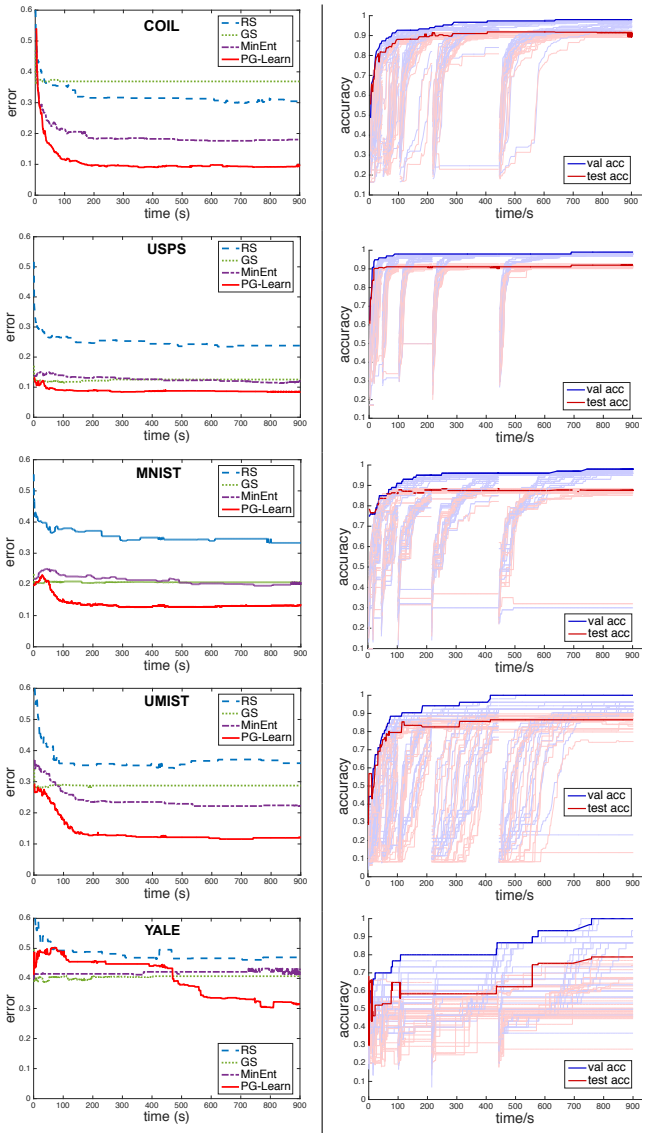
Figure 6 (a) shows how the test error changes by time for all methods on average, and (b) depicts the validation and the corresponding test accuracies for PG-LEARN on an example run. We see that PG-LEARN gradually improves validation accuracy across threads over time, and test accuracy follows closely. As such, test error drops in time. *Grid* search has a near-flat curve as it uses the same kernel bandwidth on all dimensions, therefore, more time does not help in handling noise. $Rand_d$ error seems to drop slightly but stabilizes at a high value, demonstrating its limited ability to guess parameters in high dimensions with noise. Overall, PG-LEARN outperforms competition significantly in this high dimensional noisy setting as well. Its performance is particularly noteworthy on YALE, which has small $n = 320$ but large $2d > 2K$ half of which are noise.

Finally, Figure 7 shows PG-LEARN's estimated hyperparameters, $\boldsymbol{a}_{1:d}$ and $\boldsymbol{a}_{(d+1):2d}$ (avg'ed over 10 samples), demonstrating that the noisy features $(d + 1) : 2d$ receive notably lower weights.

5 CONCLUSION

In this work we addressed the graph structure estimation problem as part of relational semi-supervised inference. It is now well-understood that graph construction from point-cloud data has

[12] All experiments executed on a Linux server equipped with 96 Intel Xeon CPUs at 2.1 GHz and a total of 1 TB RAM, using Matlab R2015b Distributed Computing Server.



(a) test error by time (b) PG-LEARN val.&test acc. by time

Figure 6: (a) Test error vs. time (avg'ed across 10 runs w/ random samples) comparing PG-LEARN with baselines on noisy datasets; (b) PG-LEARN's validation and corresponding test accuracy over time as it executes Algo. 2 on 32 threads (1 run).

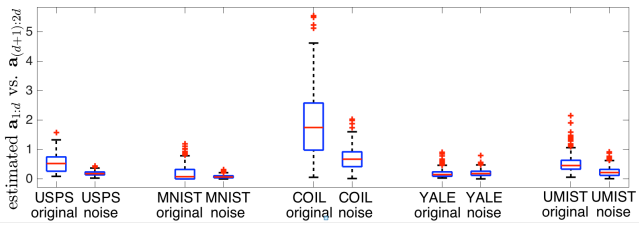


Figure 7: Distribution of weights estimated by PG-LEARN, shown separately for the original and injected noisy features on each dataset. Notice that the latter is much lower, showing it competitiveness in hyperparameter estimation.

critical impact on learning algorithms [6, 15]. To this end, we first proposed a learning-to-rank based objective parameterized by different weights per dimension and derived its gradient-based learning (§3.1). We then showed how to integrate this type of adaptive local search within a parallel framework, that early-terminates searches based on relative performance to dynamically allocate resources (time and processors) to those with promising configurations (§3.2). Put together, our solution PG-LEARN is a hybrid that strategically navigates the hyperparameter search space.

We compared PG-LEARN against existing graph estimation schemes as well as often-used standard techniques like grid search and random guessing. Experiments on a number of multi-class classification tasks both using single and parallel threads demonstrated that PG-LEARN (*i*) significantly outperforms competition, (*ii*) effectively scales up to high dimensions and (*iii*) handles noisy dimensions more effectively as compared to the baselines.

As future work we plan to deploy PG-LEARN on a distributed platform like Apache Spark, and generalize the ideas to other graph-based learning problems such as relational regression.

REFERENCES

[1] Mikhail Belkin, Partha Niyogi, and Vikas Sindhwani. 2006. Manifold Regularization: A Geometric Framework for Learning from Labeled and Unlabeled Examples. *Journal of Machine Learning Research* 7 (2006), 2399–2434.

[2] Avrim Blum and Shuchi Chawla. 2001. Learning from Labeled and Unlabeled Data using Graph Mincuts.. In *ICML*. 19–26.

[3] Bin Cheng, Jianchao Yang, Shuicheng Yan, Yun Fu, and Thomas S. Huang. 2010. Learning With L1-Graph for Image Analysis. *IEEE Tran. Img. Proc.* 19, 4 (2010), 858–866.

[4] Michael Conover, Bruno Goncalves, Jacob Ratkiewicz, Alessandro Flammini, and Filippo Menczer. 2011. Predicting the Political Alignment of Twitter Users.. In *SocialCom/PASSAT*. IEEE, 192–199.

[5] Samuel I. Daitch, Jonathan A. Kelner, and Daniel A. Spielman. 2009. Fitting a Graph to Vector Data *(ICML)*. ACM, New York, NY, USA, 201–208.

[6] Celso André R. de Sousa, Solange O. Rezende, and Gustavo E. A. P. A. Batista. 2013. Influence of Graph Construction on Semi-supervised Learning.. In *ECML/PKDD*. 160–175.

[7] Paramveer S. Dhillon, Partha Pratim Talukdar, and Koby Crammer. 2010. Learning Better Data Representation using Inference-Driven Metric Learning (IDML). In *ACL*.

[8] Lucie Flekova, Daniel Preotiuc-Pietro, and Lyle H. Ungar. 2016. Exploring Stylistic Variation with Age and Income on Twitter.. In *ACL*.

[9] G.H. Golub and C.F. Van Loan. 1989. *Matrix Computations*. Johns Hopkins University Press.

[10] Kevin G. Jamieson and Ameet Talwalkar. 2016. Non-stochastic Best Arm Identification and Hyperparameter Optimization. In *AISTATS*. 240–248.

[11] Tony Jebara, Jun Wang, and Shih-Fu Chang. 2009. Graph Construction and B-matching for Semi-supervised Learning *(ICML)*. ACM, 441–448.

[12] Masayuki Karasuyama and Hiroshi Mamitsuka. 2017. Adaptive edge weighting for graph-based learning algorithms. *Machine Learning* 106, 2 (2017), 307–335.

[13] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Efficient Hyperparameter Optimization and Infinitely Many Armed Bandits. *CoRR* abs/1603.06560 (2016).

[14] Wei Liu and Shih-Fu Chang. 2009. Robust multi-class transductive learning with graphs.. In *CVPR*. 381–388.

[15] Markus Maier, Ulrike von Luxburg, and Matthias Hein. 2008. Influence of graph construction on graph-based clustering measures.. In *NIPS*. 1025–1032.

[16] Bryan Perozzi and Steven Skiena. 2015. Exact Age Prediction in Social Networks.. In *WWW (Companion Volume)*. ACM, 91–92.

[17] Daniel Preotiuc-Pietro, Vasileios Lampos, and Nikolaos Aletras. 2015. An analysis of the user occupational class through Twitter content.. In *ACL*. 1754–1764.

[18] S.T. Roweis and L.K. Saul. 2000. Nonlinear dimensionality reduction by locally linear embedding. *Science* 290, 5500 (2000), 2323–2326.

[19] Xinhua Zhang and Wee Sun Lee. 2006. Hyperparameter Learning for Graph Based Semi-supervised Learning Algorithms.. In *NIPS*. MIT Press, 1585–1592.

[20] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. 2003. Learning with Local and Global Consistency.. In *NIPS*. MIT Press, 321–328.

[21] Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. 2003. Semi-supervised learning using Gaussian fields and harmonic functions. In *ICML*. 912–919.