Distributed Data Learning with Knowledge Transmission Topology

Yifan Li

B.S., University of California, Los Angeles, United States, 2019

A Dissertation Presented to the Graduate Faculty
of University of Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Statistics

University of Virginia
May 2024

Abstract

In today's digital era, vast amounts of data, such as hospital health records and individual device usage data, are stored in diverse locations. These distributed datasets, while essential for preserving individual privacy and managing data sizes, present unique challenges for comprehensive data analysis under the constraints arising from data sharing and aggregation. In this thesis, we investigate statistical modeling in a distributed data system along with some information transmission structures. In Chapter 2, we study a penalization-based model integration problem with a network constraint. We propose a network sparsification method that significantly reduces communication across data sites. This method is computationally more efficient while preserving estimation efficiency. In Chapter 3, we develop a Decentralized Federated Learning framework without sharing or aggregating data. We explore different knowledge-sharing mechanisms between sites, with the goal of building predictive models for each individual site without a central server. At the same time, we examine how different transmission topologies affect the efficiency of communication.

Acknowledgements

First of all, I want to extend my deepest gratitude to my advisor, Prof. Xiwei Tang. His kindness, patience, and steadfast support have been the cornerstone of my PhD journey. Prof. Tang is an exceptional mentor who has taught me more than I could have ever hoped for. Whenever I encountered obstacles in my research, he always pinpointed the problem and sparked my inspiration. He is kind and understanding, and we have always been able to talk like friends. I feel incredibly fortunate to have had him as my advisor, and I could not have completed this work without his generous guidance.

I also want to thank my committee members: Prof. Tianxi Li, Prof. Shan Yu, Prof. Zachary Lubberts and Prof. Yangfeng Ji, for their invaluable advice and support. Their insights have not only advanced my research but have also significantly contributed to my growth as a scholar. My gratitude extends to Prof. Jianhui Zhou, Prof. Karen Kafadar and Prof. Jeffrey Holt in the Statistics department for warmly welcoming me and answering all my questions about the curriculum, teaching, and career opportunities.

A heartfelt thank you goes to my family. I am deeply grateful to my parents, Mr. Xiaochao Li and Ms. Zejun Fan, whose unwavering love and encouragement have been my constant source of strength. My parents have always been my greatest fans, celebrating each of my achievements with boundless joy and enthusiasm. My father, inherently a humble man, never hesitates to express his pride when discussing my achievements with others. He speaks as if I am the greatest accomplishment of his life, his eyes lighting up and his voice filled with a warmth that underscores the deep pride he feels; I often joke that he will earn a Master's in Statistics when I get my PhD, just by virtue of his dedication to my studies. My mother has always been

a calming presence in my life. Her voice, soothing and wise, reassures me that no matter the distance or challenge, I am never truly alone. From the day I left home at age 15, despite being thousands of miles apart, they have made sure that home is never too far away. I would like to thank my uncle and aunt, Mr. Guofang Cheng and Ms. Xiaoli Li, for taking care of me when I went to school in Canada. They treated me as their own child, and to this day, I still see Toronto as my second home. I want to thank my cousin Victor Cheng for his unfailing help and companionship during my 12 years of studying in the U.S. and Canada. He meticulously proofread every one of my essays and assisted me with all my school applications. It fills me with pride to see him now on his path to becoming a lawyer. I want to thank my grandma, Ms. Shuying Gong, who used to carry me on her back wherever she went when I was a toddler. She raised me, taught me how to read and write, and lived with me until middle school age. Her love and faith in me have profoundly shaped the person I am today.

Thank you to my peers, Mingyu Qi, Yuda Shao, James Lee and Noah Gade, for their companionship and support throughout the journey. I am equally grateful to Leo Liao, Cheng Chen, Xu Ouyang, Renny Li and many other friends for the countless moments of laughter and conversation that lightened the days. A special thank you to Evelyn Tsai, whose presence has been a constant source of joy and inspiration during the final year of my PhD.

I've been picturing this moment since the first day of my PhD: completing all the requirements for my degree, with only this acknowledgment left to write. I thought five years would feel long, but looking back, so many moments stand out vividly. I remember arriving at Charlottesville's Amtrak Station, initially surprised by its quiet charm compared to the bustling cities I was used to, but soon falling in love with it. My first day at Halsey Hall, finding my office on the ground floor

in the corner of the building at B003, was not in a fancy office, but I immediately found my sense of belonging there. The summer I prepared for my qualifying exam was intense; I grinded nonstop for two months, tackling practice problems and memorizing equations. I passed on my first try, although I took it at least ten times in my dreams, and still think it was one of the hardest exams I've ever faced. My first day teaching as a lecturer, I was so nervous that I spoke too fast and finished the lecture in 20 minutes. Receiving Christmas cards and reading comments from my students, knowing they enjoyed my class and actually learned from it, gave me a great sense of achievement. One year later, I was nominated for a University teaching award. I remember playing basketball alone at Jefferson Ridge when I received an internship offer from Amazon—I initially thought it was a rejection letter, but it began with congratulations. The day of my thesis defense, April 11, 2024, which is exactly five years after I committed to UVA, was when it truly hit me how quickly time flies. My time at UVA has been enjoyable and unforgettable, undoubtedly the best time of my life.

Lastly, I am grateful for all the grinding, sleepless nights, the highs and lows, the moments of joy and the moments of doubt. They have all taught me the value of perseverance, showing me that resilience is forged through adversity. My time at UVA has been not just a phase of academic growth, but the most transformative and enriching period of my life, filled with lessons I will proudly carry forward in the future.

Contents

Abstract iii

Acknowledgements iv

1 Introduction **1**

1.1 Distributed Data 2

1.2 Centralized and Decentralized Schemes 4

1.3 Outline 6

2 Model Aggregation Under Network-Subjected Constraint **8**

2.1 Introduction 8

2.2 Network-Induced Optimization 12

2.2.1 Background and Notation 12

2.2.2 Fusion with ℓ_1 14

2.2.3 Sparsified Network Constraint 15

2.3 Commonly Adopted Algorithms 17

2.3.1 Subgradient Descent 17

2.3.2 Coordinate Descent 18

2.3.3 Alternating Direction Method of Multipliers (ADMM) 19

2.4 Network Sparsification 21

2.4.1 Naive Sparsifiers 21

2.4.2 Proposed Sparsifiers 27
2.4.3 Weighted Sparsifiers 30
2.5 Simulations 33
2.5.1 Evaluation Metric 33
2.5.2 Comparison on Different Graphs 34
2.5.3 Sparsifiers with Different Density 40
2.5.4 Centrality of Graphs 45
2.5.5 Weighted vs. Unweighted Sparsifiers 45
2.5.6 Comparison with Community Detection 47
2.6 Theoretical Results 50
2.7 Application on Email Data 57
2.8 Discussion 59

3 Decentralized Federated Learning and Knowledge Transfer 61
3.1 Introduction 61
3.2 Notations 68
3.3 Decentralized Federated Learning 69
3.3.1 Local Estimation 69
3.3.2 Data Aggregation 70
3.3.3 Sharing Gradient 71
3.3.4 Model Fusion 72
3.3.5 Mutual Knowledge Transfer 73
3.4 Optimization Approaches 79
3.4.1 Gradient Descent 79
3.4.2 Stochastic Gradient Descent (SGD) 80
3.4.3 Fixed and Decay Step Size 82

3.4.4 Adaptive Moment Estimation (ADAM) Optimization 83
3.5 Transmission Topology 85
3.5.1 Fully Connected Network 85
3.5.2 Chain Network 86
3.5.3 Star Network 87
3.6 Toy Example 89
3.7 Application on MNIST Data 90
3.7.1 Binary Classification with Distribution Shift 91
3.7.2 Classification with Partial Information 91
3.7.3 Classification with Multiple Sites 93
3.7.4 Network Efficiency 95
3.8 Theoretical Results 97
3.9 Discussion 101

Bibliography **102**

Chapter 1

Introduction

Data is omnipresent and omnipotent in our daily lives, with countless amounts being collected every second in various forms. As a result, big data is gradually becoming a hot topic in data analysis and Machine Learning. Many services rely on large amounts of user data to make inferences, such as e-health, advertising, and smart cities. With this data, Deep Learning algorithms are able to provide better and more advanced services in voice-based personal assistants, self-driving cars, image processing, and more.

Traditionally, vast amounts of data are collected into a centralized cloud server, and Machine Learning models import data from the server during the training process. However, the rapid development of the online industry and advertising have fueled the need to store personal data where it is collected. For example, Internet of Things (IoT) devices, mobile phones, and medical data are stored in distributed resources. The distributed nature of these data sources presents several challenges for traditional storage and analysis methods, including privacy concerns, data security, and logistical difficulties, such as transferring large volumes of data to a centralized location for processing.

Zerka et al. (2020) discuss the potential of distributed learning as a solution in healthcare to maintain the privacy of patient data. Their work highlights the promise of distributed algorithms in enabling learning from isolated datasets without the need to share the data itself. Xu et al. (2018) propose a game-theoretical approach to study how users choose their privacy budget to achieve a balance between the accuracy of a classifier and preserved privacy. Based on differential privacy, they propose a Distributed Differential Privacy protocol for private data aggregation, which generates differentially private aggregate results from distributed databases. Jena et al. (2013) aim to find frequent itemsets in a distributed database without revealing data to others. They propose a data mining privacy by decomposition method that uses a genetic algorithm to search for the optimal feature set.

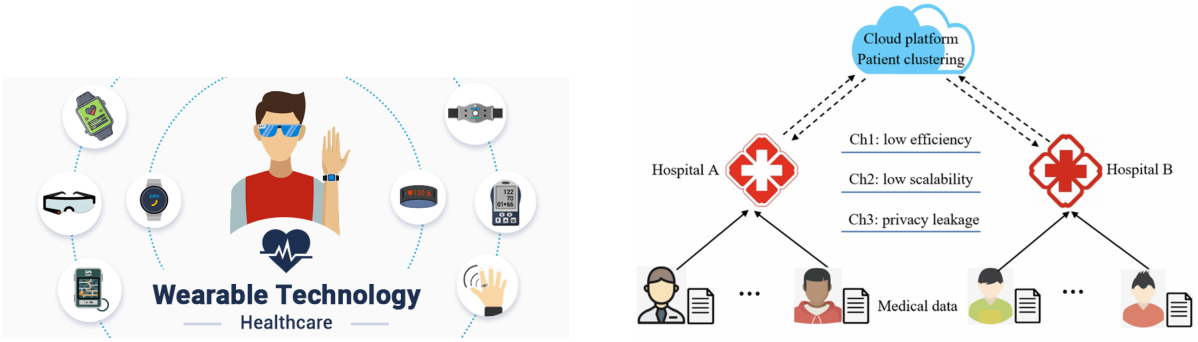
1.1 Distributed Data

Distributed data involves the placement of data across various locations to optimize processing and storage efficiencies. Compared to centralized systems, distributed systems leverage multiple locations to manage and process data, which not only enhances data accessibility and processing speed but also improves the system's robustness against failures and data loss. Distributed data architectures have become a cornerstone of modern data management strategies. By storing and processing data closer to its source, these architectures minimize latency, enabling more responsive and scalable systems (Martínez Beltrán et al., 2022; El-Sayed et al., 2023; Panigrahi et al., 2023).

The main benefit of distributed data is that it protects user privacy by keeping data locally. In current literature, cryptographic computations are common privacy-preserving algorithms for distributed data. For instance, Jha et al. (2005)

presents homomorphic encryption protocols for the privacy-preserving computation of cluster means. Cho et al. (2018) introduce a protocol for large-scale genome-wide analysis that aids in quality control while preserving the confidentiality of underlying genotypes and phenotypes, allowing individuals to contribute their genomes without compromising privacy. Gilad-Bachrach et al. (2016) proposes CryptoNets, neural networks that can be applied to encrypted data. This method not only allows data to be sent in encrypted form but also ensures that predictions are returned to owners privately. Additionally, Gentry (2009) proposes a homomorphic encryption scheme, which allows people to evaluate encrypted data without needing to decrypt it.

Another way of preserving privacy is through randomization, also known as differential privacy. Dwork (2006) explains the concept of differential privacy and how it has developed over time. Abadi et al. (2016) develops an algorithm for Deep Learning within the framework of differential privacy, which maintains a decent privacy budget without sacrificing model quality and training efficiency. Dwork et al. (2006) prove that privacy can be preserved by adding noise to the data, where the standard deviation of the noise is correlated to the sensitivity of the function.



(a) Wearable technology data (b) Hospital data

Figure 1.1: Distributed data examples.

1.2 Centralized and Decentralized Schemes

There are two schemes to analyze distributed data: centralized analysis and decentralized analysis. The debate between centralized and decentralized data analysis approaches is driven by the trade-offs between data governance, performance, and privacy protection. Centralized systems are simpler in data management; essentially, all the data is sent to a central server, which is potentially more powerful in computational resources. Centralized analysis uses all available data points from the entire dataset to estimate parameters, offering a comprehensive overview of the parameter behavior across all data. It is preferable when we wish to have a broad picture of parameter behavior across all data. Centralized analysis is the traditional approach for Machine Learning in many different areas. For instance, Zhao et al. (2008) provides a comprehensive analysis of sensor data analysis, which involves gathering data from different sensors and analyzing it using various regression methods. Manglik et al. (2023) provides an overview of Machine Learning-based road safety prediction techniques. They show that most traditional analyses on road traffic, vehicle hindrances estimation, and vehicle safety measure prediction are centralized, but more concerns have arisen in recent years regarding data storage and privacy. In agriculture, Liakos et al. (2018) demonstrate the wide application of traditional Machine Learning algorithms such as regression and classification in crop management, stock management, water management, and soil management. All these problems involve sending sensor data to a central location and analyzing them together. In healthcare, body sensors track a person's activity, send it to remote cloud storage, and then process it to check if a medical emergency is required (Pandey et al., 2022). In transportation, researchers use Machine Learning to analyze traffic data collected from multiple sources such as GPS or road sensors to improve traffic flow

forecasting, incident processing, and autonomous driving (Nguyen et al., 2018).

A common problem pointed out by these papers is the privacy concern. It is hard to protect user privacy as all the information is shared across sites. This concern is supported by works published previously. Baruh et al. (2017) explores the privacy concerns of online services and social network sites from information sharing and proposes corresponding privacy measures. Tsarenko and Tojib (2009) explore public concerns about privacy in financial institutions and examine the driving factors. Turn et al. (1976) compare the privacy threats of centralized and decentralized databank systems and concludes that a properly designed centralized system can better protect privacy. At the same time, centralized analysis is susceptible to single points of failure. If one element of the system malfunctions for some reason, the entire system can be disrupted or shut down. For instance, Baladi and Hendrix (2011) shows that a single point of failure of equipment at the Department of Energy Savannah River site can cause unacceptable schedule delays. Sun (2009) compares centralized and decentralized management in software systems and concludes that centralized management is more vulnerable, despite slightly lower costs.

Decentralized analysis, on the other hand, leverages distributed data without the need for aggregation, thus preserving privacy and reducing the risk of data failures. At the same time, in scenarios where localized patterns are crucial, decentralized learning models are preferred due to their sensitivity to local nuances (Ferrag et al., 2021; Liu et al., 2023; Schumann et al., 2023). However, it is harder to aggregate information across different sites, and local models can be biased if local data is not representative of the population. The decentralized approach outperforms the centralized one in many cases. For example, Allende-Cid et al. (2016) compare a centralized approach and two distributed approaches for various distributed regression tasks on synthetic and real data. The results indicate that distributed approaches

perform better in most cases. Xin et al. (2020) review decentralized stochastic first-order methods on large-scale Machine Learning tasks. They present a unified framework that combines variance reduction with gradient tracking to achieve better performance and convergence. Aketi et al. (2023) propose a Decentralized Federated Learning algorithm called Global Update Tracking (GUT), which aims to reduce the impact of heterogeneous data in decentralized learning without introducing any communication overhead. Talistu et al. (2015) propose a distributed data analysis method that uses hierarchical clustering for local online summaries. At the same time, they utilize a gossip protocol for the summaries and spectral clustering for offline analysis. Compared to centralized analysis, the resulting solution significantly reduces communication overhead and is computationally less expensive.

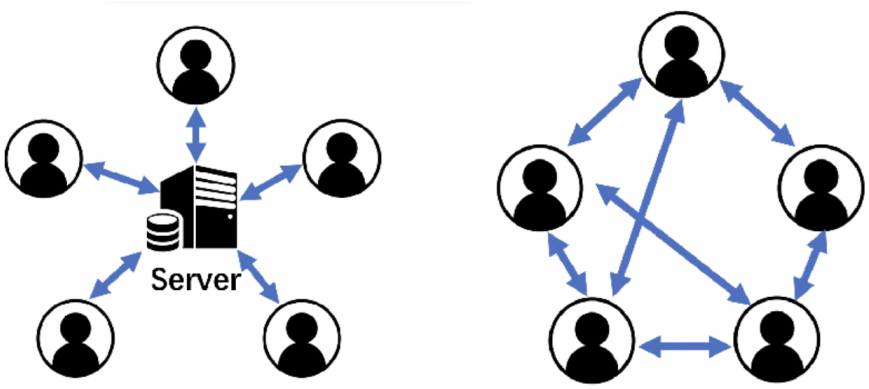


Figure 1.2: Centralized vs. Decentralized analysis.

1.3 Outline

This thesis aims to explore the centralized and decentralized approaches to distributed data in depth. For the first project, we study a penalization-based model

integration problem with a network constraint. Specifically, we explore different ways to generate sparsified graphs based on the original network. These sparsified graphs have far fewer edges, but can solve the regression problem much more efficiently without sacrificing too much prediction accuracy. The second project focuses on a Decentralized Federated Learning problem. Instead of viewing the entire network as a whole, we take the analysis to each site. Given that each site has its own data, we explore various ways to transmit knowledge to other sites while protecting the privacy of each individual site. Additionally, we investigate the effect of different network mechanisms on knowledge sharing efficiency. Assuming that all sites are interconnected, constructing the communication network differently will significantly affect the communication speed.

Chapter 2

Model Aggregation Under Network-Subjected Constraint

2.1 Introduction

In distributed data learning problems with multiple sites, local models often suffer from limited data size. Training a Machine Learning model on a dataset that is inherently small or limited significantly impacts its ability to learn and generalize. Each individual dataset may exhibit very low bias, but the small sample size introduces a high variance issue. In simpler terms, our model might perform exceptionally well on the training data but fail to generalize to new data. Ideally, we would want our models to have both low bias and low variance, meaning they accurately predict unseen data and perform consistently across different data samples. However, achieving this balance in practice is challenging. High bias can lead to underfitting, where the model is too simplistic, and high variance can lead to overfitting, where the model captures noise in the training data as if it were a real pattern. Our proposed solution to this dilemma is model aggregation. By aggregat-

ing models trained on "similar" data sites, we can significantly reduce variance while introducing some bias. This approach is particularly effective when local data are considered IID (Independently and Identically Distributed), meaning each dataset is a random, equally representative sample from the overall population. The new question then arises: how do we determine which of the sites are similar to each other? This is where network structure comes into play.

In some problems, we have a network structure as well as observed values on each node. If the goal is to learn global signals, one common assumption is to assume signal similarities and aggregate some of the models together. Solving a statistical estimation problem on a graph is a popular problem. Specifically, we want to estimate the signals on the graph using observed values and graph information. Under the regression framework, a series of penalized regression models have been developed and incorporated into graph problems. Methods including the Least Absolute Shrinkage and Selection Operator (LASSO)(Tibshriani, 1996), Smoothly Clipped Absolute Deviation (SCAD)(Fan and Li, 2001), Fused LASSO (Tibshirani et al., 2005), and Elastic Net (Zou and Hastie, 2005) have been adapted to solve graph learning problems after being proposed. For instance, Li and Li (2008) proposed solving a regression problem on genomics data using a network-constrained ℓ_2 regularization. Sun and Li (2010) proposed a Bayesian approach to a graph-constrained regression problem.

One motivating example of this paper comes from a New York City taxi trip problem in Wang et al. (2016b). The goal is to detect events based on abnormalities in the number of taxi trips at different locations in New York City. Each junction is a node, and junctions are connected if there exists a road. In this example, there are 3874 nodes (junctions) and 7070 edges (sections of roads that connect two junctions). The observed value is the difference between the counts observed during

the Gay Pride parade in a two-hour window on June 26, 2011, and the seasonal averages. Figure 2.1 shows the true parade route and unfiltered signal. The goal is to recover the true signal using the noisy graph. From the result, we can see that Sparse trend filtering recovers the signal better than Laplacian smoothing, as it can better localize the estimate when there are strong spikes in the measurement.

However, when the graph is dense, computation becomes an issue as the computational cost increases exponentially with the dimension. In previous literature, improvements have been made on the regression side. Specifically, for Fused LASSO, numerous algorithms have been proposed to speed up the computation process. Davies and Kovac (2001) derive a taut-string multi-resolution method that solves the 1D Fused LASSO problem, which is later extended by Condat (2012) and Barbero and Sra (2014). Johnson (2013) propose a new approach for the Fused LASSO based on dynamic programming. The taut-string and dynamic programming algorithms are very fast and practical in use. Subsequently, these methods have been applied to graph-based problems. Kolmogorov et al. (2016) extend the dynamic programming approach of Johnson (2013) to solve the Fused LASSO problem on a tree. Barbero and Sra (2014) extend the fast 1D Fused LASSO optimizer to work over grid graphs, using operator splitting techniques like Douglas Rachford splitting. Hoefling (2010) and Tibshirani and Taylor (2011) propose solution path algorithms; Tansey and Scott (2015) leverage fast 1D Fused LASSO solvers in an ADMM decomposition over trails of the graph; Landrieu and Obozinski (2015) derive a new method based on graph cuts. The problem with these techniques is that although they are lightning-fast on specific types of graphs, they are still not so efficient for arbitrary graphs.

Therefore, this leads to the idea of simplifying the problem on the graph side. Specifically, we can sparsify the graphs so that there are fewer edges, which reduces

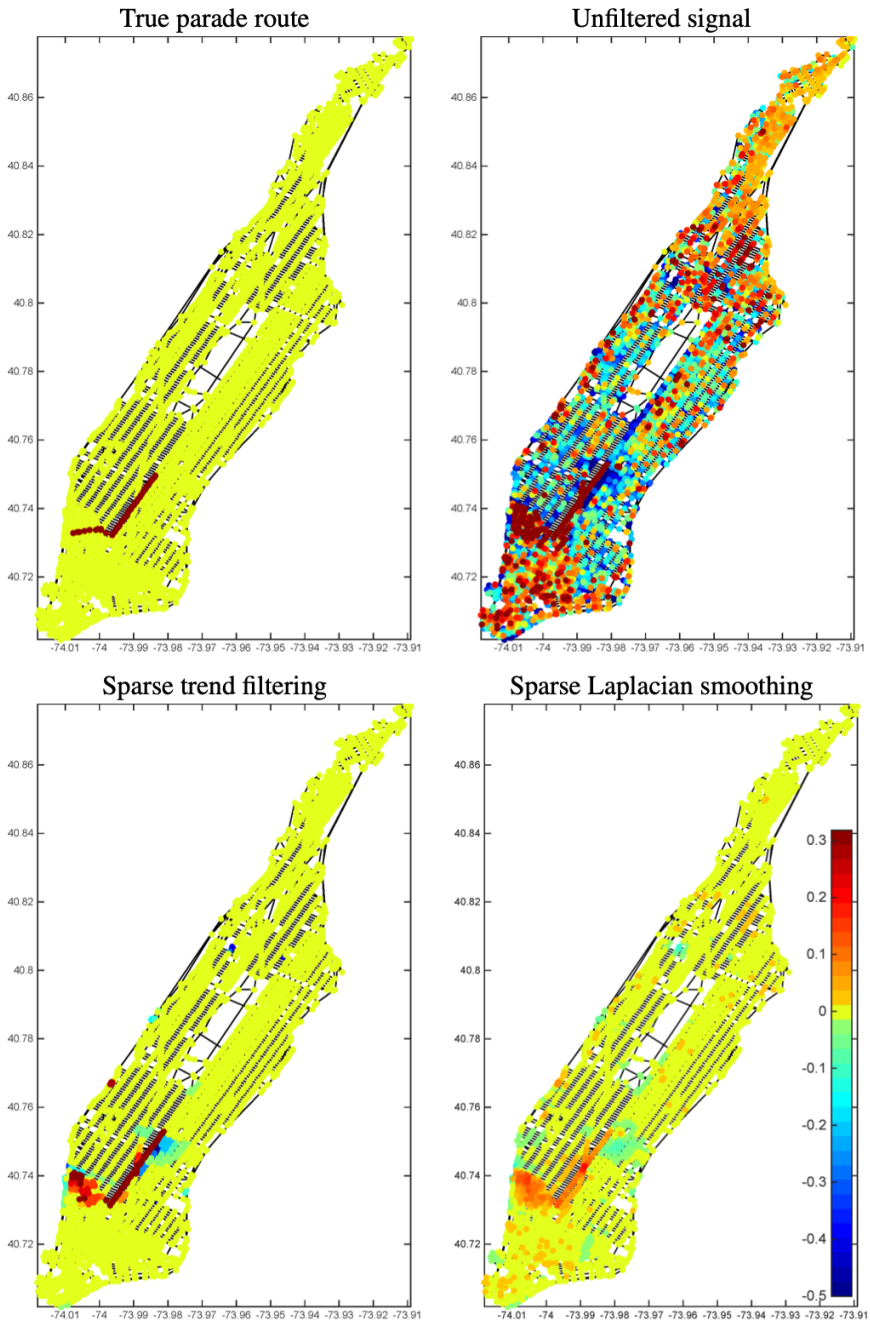


Figure 2.1: A graph-constrained problem and its estimated signals.

the dimension of the incidence matrix, and therefore increases the computation speed. Currently, there is very little literature on this topic. The motivation for this

paper came from Sadhanala et al. (2016). In this paper, the authors proposed several methods to solve an ℓ_2 penalization problem on a dense graph. Specifically, since the result of ℓ_2 penalization is separable, there is an explicit solution for the true signal. A natural question to ask is: what if we have ℓ_1 penalization? By nature, ℓ_1 penalized regression does not have an explicit solution and therefore cannot be estimated explicitly. In this paper, we aim to solve a Fused LASSO problem on a dense graph efficiently and accurately by sparsifying the graph using various methods. After sparsification, computation time significantly decreases, without sacrificing too much accuracy. We proposed some new sparsification methods including K Nearest-neighbor (KN) based sparsifier and Random Walk (RW) sparsifier. The results of our study show that KN has the best performance in almost all scenarios. Also, we are able to adjust the number of edges for three methods by constructing the trees in parallel or sequentially, which adds flexibility to the algorithm.

2.2 Network-Induced Optimization

2.2.1 Background and Notation

Suppose there are distributed data sources with data collections $(y_1, x_1), (y_2, x_2), \ldots, (y_n, x_n)$. At each data site, consider a learning function $f(y_i, x_i; \beta_i)$. Then the objective function is

$$\arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} f(y_i, x_i; \beta_i) + \lambda p(\boldsymbol{\beta}), \tag{2.1}$$

where $f(y_i, x_i; \beta_i)$ can be any loss function at each node, $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_n)^T$, $p(\boldsymbol{\beta})$ is a network-induced penalization to encourage information sharing across different sites, and λ is a tuning parameter.

There are two common methods for penalization on network-constrained prob-

lems: Trend Filtering (Wang et al., 2016a) or Laplacian Smoothing (Smola and Kondor, 2003). The idea is to penalize the pairwise difference between the magnitudes of coefficients. Trend Filtering penalizes the absolute difference between coefficients, and Laplacian Smoothing penalizes the square of the difference.

Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be an undirected graph with vertex set $\boldsymbol{V} = \{1, 2, \ldots, n\}$, and edge set $\boldsymbol{E} \subseteq \{(i, j) : i, j \in \boldsymbol{V}\}$ of size $m = |\boldsymbol{E}|$. Let $\boldsymbol{\beta} = \{\beta_i : i \in \boldsymbol{V}\}$ be a signal whose components are associated with the vertices of $\boldsymbol{G}$. Our goal is to estimate $\boldsymbol{\beta}$ on the basis of noisy observations $\boldsymbol{y}$. In general, the overall complexity of the problem has two distinct parts: local complexity and communication cost. Since our study focuses on global optimization with respect to network constraints, which is independent of local model complexity, without loss of generality, we consider a simple case for illustration, which is an intercept model $y_i = \beta_i + \epsilon_i$.

The ℓ_2 fusion method is one of the most common ones. The idea is to penalize the square of the difference between coefficients and force them to shrink together. Given observations $\boldsymbol{y} = (y_1, \ldots, y_n) \in Y^n$ over nodes of $\boldsymbol{G}$ and for weights w_{ij}, $(i, j) \in \boldsymbol{E}$, the objective function is

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \boldsymbol{\beta}^T \boldsymbol{L} \boldsymbol{\beta} = \min_{\boldsymbol{\beta} \in \mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \sum_{(i,j) \in \boldsymbol{E}} w_{ij} (\beta_i - \beta_j)^2,$$

where $\sum_{i=1}^{n} f(y_i, \beta_i) = \sum_{i=1}^{n} ||y_i - \beta_i||_2^2$. Recall that the Laplacian $\boldsymbol{L} \in \mathbb{R}^{n \times n}$ of $\boldsymbol{G}$ is

$$L_{ij} = \begin{cases} \sum_{(i,l) \in E} w_{il} & \text{if } i = j \\ -w_{ij} & \text{if } i \neq j \end{cases}, i, j \in \boldsymbol{V}.$$

Therefore, the solution can be written in explicit form:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{I} + \lambda \boldsymbol{L})^{-1} \boldsymbol{y}.$$

However, there are two potential problems with this approach. First of all, the solution involves a large matrix inverse, which can be computationally expensive when the dimension is high. Also, since ℓ_2 fusion penalizes the squared difference among the coefficients, which is similar to Ridge regression, the procedure does not separate a sparse solution in terms of a large number of pairwise differences. To better identify the pattern of distributed signals under the connecting structure, an alternative choice is to employ an ℓ_1 penalization for fusion on the piece-wise difference, instead of ℓ_2 penalization. However, ℓ_1 penalization is computationally more challenging since there is no explicit solution.

2.2.2 Fusion with ℓ_1

The idea of ℓ_1 fusion comes from Fused LASSO (Tibshirani et al., 2005). The aim is to penalize the absolute difference between coefficients and force them to be the same, for coefficients with similar values. For ℓ_1 fusion, the objective function is

$$\hat{\boldsymbol{\beta}} = \min_{\boldsymbol{\beta}\in\mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \sum_{(i,j)\in \boldsymbol{E}} w_{ij}|\beta_i - \beta_j|, \tag{2.2}$$

where $\sum_{i=1}^{n} f(y_i, \beta_i) = \sum_{i=1}^{n} \|y_i - \beta_i\|_2^2$.

If for each edge $e \in \boldsymbol{E}$, we arbitrarily select one of the two joined vertices to be the head, denoted e^+, and the other to be the tail, denoted e^-. Then, we define a row $(\boldsymbol{\nabla_G})_e$ of $\boldsymbol{\nabla_G}$, corresponding to the edge e, by

$$(\boldsymbol{\nabla_G})_{e,e^+} = 1, (\boldsymbol{\nabla_G})_{e,e^-} = -1, (\boldsymbol{\nabla_G})_{e,v} = 0 \text{ for all } v \neq e^+, e^-, \tag{2.3}$$

for each $e \in \boldsymbol{E}$. For an unweighted graph, we can write the objective function in

Equation 2.2 as

$$\hat{\boldsymbol{\beta}} = \min_{\boldsymbol{\beta}\in\mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \|\boldsymbol{\nabla}_{\boldsymbol{G}}\boldsymbol{\beta}\|_1. \tag{2.4}$$

For an arbitrary $\boldsymbol{\beta}$, we have

$$\|\boldsymbol{\nabla}_{\boldsymbol{G}}\boldsymbol{\beta}\|_1 = \sum_{e\in\boldsymbol{E}} |\beta_{e^+} - \beta_{e^-}|.$$

Therefore, the objective function becomes:

$$\hat{\boldsymbol{\beta}} = \min_{\boldsymbol{\beta}\in\mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \sum_{e\in\boldsymbol{E}} |\beta_{e^+} - \beta_{e^-}|.$$

However, there is no explicit solution for this problem due to the nature of ℓ_1 penalization.

2.2.3 Sparsified Network Constraint

When solving a Network-induced Fused LASSO problem, the number of edges grows quadratically with respect to the number of nodes, as shown in Figure 2.2. This implies that when the network is large and dense, solving the estimation problem becomes computationally expensive. Our proposed solution involves sparsification on the graph side. We aim to simplify the graph so that it contains the same number of nodes, but with far fewer edges. Ideally, the edges in the sparsified graph should capture all of the true relationships between the nodes, with very little noise being captured. In this way, we can ensure that we maintain the graph structure while minimizing the computation required.

Recall that $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is an undirected graph with vertex set $\boldsymbol{V} = \{1, 2, \ldots, n\}$, and edge set $\boldsymbol{E} \subseteq \{(i, j) : i, j \in \boldsymbol{V}\}$. Let $\tilde{\boldsymbol{G}} = (\boldsymbol{V}, \tilde{\boldsymbol{E}})$ be a sparsified graph of $\boldsymbol{G}$

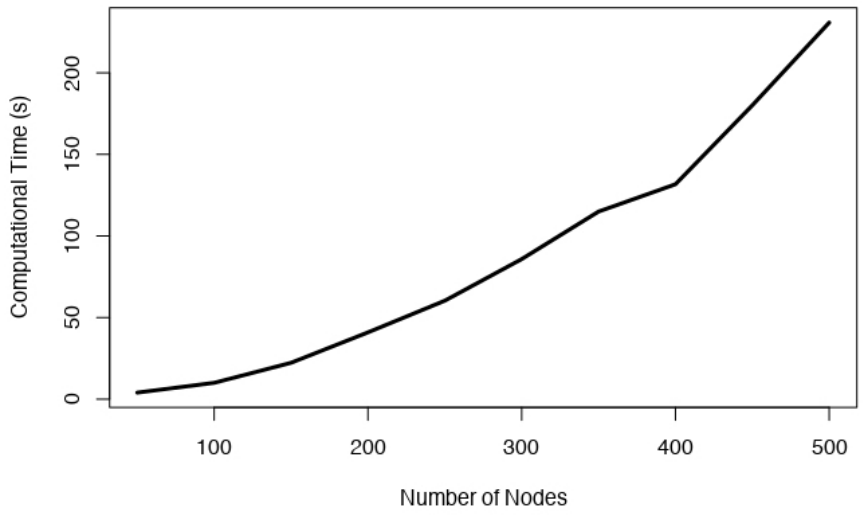


Figure 2.2: Computational time for Network-induced Fused LASSO problems.

with $|\tilde{\boldsymbol{E}}| < |\boldsymbol{E}|$. Then the corresponding objective function is

$$\tilde{\boldsymbol{\beta}} = \min_{\boldsymbol{\beta} \in \mathbb{R}^n} \sum_{i=1}^{n} f(y_i, \beta_i) + \lambda \|\boldsymbol{\nabla}_{\tilde{\boldsymbol{G}}} \boldsymbol{\beta}\|_1.$$

Overall, our goal is to ease the computational burden while maintaining overall accuracy when solving the optimization problem. There are different factors that we want to explore in this paper. First of all, we explore several different kinds of sparsifiers. Additionally, for a sparsifier, we want to explore the different possible ways to construct it, including both weighted and unweighted methods. We also explore how the density of the sparsified graph affects the estimation result. We test the sparsifiers on different kinds of graphs, which have various grouping structures and densities. Lastly, we compare the sparsifiers with Community Detection algorithms. We show that the sparsifiers can outperform Community Detection algorithms under certain graph structures and conditions.

2.3 Commonly Adopted Algorithms

2.3.1 Subgradient Descent

Subgradient Descent is designed based on the traditional Gradient Descent algorithm with a slight modification in the updating step. It is intended to handle non-differentiable functions and is particularly useful for optimizing convex functions that are not differentiable everywhere. Instead of the gradient, Subgradient Descent uses a subgradient, which is a generalization of the gradient for non-differentiable points. This method allows the optimization process to proceed even without a well-defined slope, and a well-known example is the derivative of the absolute value, which is not differentiable at 0. Similarly, Subgradient Descent is a powerful tool for solving optimization problems involving hinge loss and other piece-wise linear functions. These loss functions are common in Machine Learning, especially in classification tasks and regularization techniques.

The update rule for Subgradient Descent at iteration k can be written as:

$$\boldsymbol{\beta}^{(k+1)} = \boldsymbol{\beta}^{(k)} - \alpha^{(k)}\boldsymbol{g}^{(k)},$$

where $\boldsymbol{\beta}^{(k)}$ is the parameter vector at iteration k, $\alpha^{(k)}$ is the step size at that iteration, and $\boldsymbol{g}^{(k)}$ is a subgradient of the function f at $\boldsymbol{\beta}^{(k)}$. Here, $\boldsymbol{g}^{(k)} \in \partial f(\boldsymbol{\beta}^{(k)})$, with $\partial f(\boldsymbol{\beta}^{(k)})$ representing the subdifferential of f at $\boldsymbol{\beta}^{(k)}$. The subdifferential is defined as the set of vectors $\boldsymbol{g}$ that satisfy the following condition for all vectors $\boldsymbol{\beta}$:

$$f(\boldsymbol{\beta}) \geq f(\boldsymbol{\beta}^{(k)}) + \boldsymbol{g}^T(\boldsymbol{\beta} - \boldsymbol{\beta}^{(k)}).$$

This ensures that $\boldsymbol{g}$ points in a direction where the function f does not decrease,

providing a way to generalize the concept of direction of ascent or descent for non-differentiable functions. Algorithm 1 shows the pseudo code for Subgradient Descent.

Algorithm 1 Subgradient Descent

procedure SUBGRADIENTDESCENT(f, α)
    Input: Objective function f, learning rate α
    Output: Optimized parameters $\boldsymbol{\beta}$
    Initialize parameters $\boldsymbol{\beta}$
    while convergence criteria not met **do**
        Compute a subgradient $\boldsymbol{g}$ at $\boldsymbol{\beta}$, where $\boldsymbol{g} \in \partial f(\boldsymbol{\beta})$
        Update $\boldsymbol{\beta}$: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \cdot \boldsymbol{g}$
    end while
    return $\boldsymbol{\beta}$
end procedure

2.3.2 Coordinate Descent

Coordinate Descent is a common optimization algorithm in Machine Learning literature. The basic idea of it is to convert a high-dimensional problem into one-dimensional, by optimizing along one direction while fixing others at each iteration step. This algorithm picks a direction to optimize, and runs iteratively until convergence. This approach is particularly effective for high-dimensional problems where optimization with respect to one coordinate can be performed more efficiently than in the full-dimensional space. Coordinate Descent has been proved successful in various areas of Machine Learning, including sparse modeling and high-dimensional data analysis. Its simplicity and efficiency make it one of the go-to algorithms when dealing with large-scale optimization problems.

Given a function $f(\boldsymbol{\beta})$ where $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_n) \in \mathbb{R}^n$, the update for the i-th coordinate in the k-th iteration can be written as:

$$\beta_i^{(k+1)} = \arg\min_y f(\beta_1^{(k+1)}, \ldots, \beta_{i-1}^{(k+1)}, y, \beta_{i+1}^{(k)}, \ldots, \beta_n^{(k)}),$$

where $\beta_i^{(k)}$ is the value of the i-th coordinate at the k-th iteration. Algorithm 2 shows the pseudo code for Coordinate Descent.

Algorithm 2 Coordinate Descent

procedure CoordinateDescent($f, \boldsymbol{\beta}$)
  Input: Objective function f, initial parameters $\boldsymbol{\beta} = (\beta_1, \beta_2, \ldots, \beta_n)$
  Output: Optimized parameters $\boldsymbol{\beta}$
  Initialize parameters $\boldsymbol{\beta}$
  while convergence criteria not met **do**
    for $i = 1$ to n **do**
      Define $g(\beta_i) = f(\beta_1, \ldots, \beta_{i-1}, \beta_i, \beta_{i+1}, \ldots, \beta_n)$
      $\beta_i \leftarrow \arg\min_y g(y)$ where y replaces β_i in g
      Update β_i to the value that minimizes $g(y)$
    end for
  end while
  return $\boldsymbol{\beta}$
end procedure

2.3.3 Alternating Direction Method of Multipliers (ADMM)

The Alternating Direction Method of Multipliers (ADMM) is an optimization algorithm that effectively solves a wide range of problems in Machine Learning. This method was first proposed by Boyd (2010) and quickly gained popularity in Machine Learning literature. The idea is to decompose the problem into smaller, more tractable subproblems, which are easier to optimize. Using augmented Lagrangian methods and dual decomposition, ADMM iteratively updates guesses for the solution for the subproblems and eventually leads to the goal of optimizing the desired variables.

An important part of the ADMM algorithm is its parameters and variables, which

include the convex functions $f(\boldsymbol{\beta})$ and $g(\boldsymbol{z})$ for minimization. The matrices $\mathbf{A}$ and $\mathbf{B}$ connect the variables $\boldsymbol{\beta}$ and $\boldsymbol{z}$ to the constraint equation defined by the vector $\mathbf{c}$. The penalty parameter ρ is a tuning parameter that controls the convergence rate. The dual variable $\boldsymbol{u}$ is the Lagrange multiplier that enforces the constraint.

Here are the three updating steps for ADMM: The update for $\boldsymbol{\beta}$ aims to minimize the objective $f(\boldsymbol{\beta})$ with a quadratic penalty. This term penalizes any deviation from the specified constraint, which shrinks $\boldsymbol{\beta}$ toward compliance. Similarly, the $\boldsymbol{z}$ update optimizes the objective $g(\boldsymbol{z})$, pushing $\boldsymbol{z}$ toward the latest $\boldsymbol{\beta}$ values with a penalty term associated with constraint violation. The dual variable $\boldsymbol{u}$ is updated to show the current level of discrepancy from the constraint. At the same time, this update directs subsequent iterations towards meeting the constraints. Below are the updates in equation form:

$$
\begin{aligned}
\boldsymbol{\beta}^{k+1} &\leftarrow \arg\min_{\boldsymbol{\beta}} \left(f(\boldsymbol{\beta}) + \frac{\rho}{2} \|\mathbf{A}\boldsymbol{\beta} + \mathbf{B}\boldsymbol{z}^k - \mathbf{c} + \boldsymbol{u}^k\|_2^2 \right), \\
\boldsymbol{z}^{k+1} &\leftarrow \arg\min_{\boldsymbol{z}} \left(g(\boldsymbol{z}) + \frac{\rho}{2} \|\mathbf{A}\boldsymbol{\beta}^{k+1} + \mathbf{B}\boldsymbol{z} - \mathbf{c} + \boldsymbol{u}^k\|_2^2 \right), \\
\boldsymbol{u}^{k+1} &\leftarrow \boldsymbol{u}^k + \mathbf{A}\boldsymbol{\beta}^{k+1} + \mathbf{B}\boldsymbol{z}^{k+1} - \mathbf{c}.
\end{aligned}
$$

Algorithm 3 shows the pseudo code for ADMM.

Algorithm 3 Alternating Direction Method of Multipliers (ADMM)

procedure ADMM($f, g, \mathbf{A}, \mathbf{B}, \mathbf{c}, \rho$)
  Input: Convex functions f and g, matrices $\mathbf{A}$ and $\mathbf{B}$, vector $\mathbf{c}$, scalar ρ
  Output: Optimized variables $\boldsymbol{\beta}$, $\boldsymbol{z}$, and dual variable $\boldsymbol{u}$
  Initialize $\boldsymbol{\beta}^0$, $\boldsymbol{z}^0$, $\boldsymbol{u}^0$
  Set iteration counter $k \leftarrow 0$
  repeat
    $\boldsymbol{\beta}^{k+1} \leftarrow \arg\min_{\boldsymbol{\beta}} \left(f(\boldsymbol{\beta}) + \frac{\rho}{2} \|\mathbf{A}\boldsymbol{\beta} + \mathbf{B}\boldsymbol{z}^k - \mathbf{c} + \boldsymbol{u}^k\|_2^2 \right)$
    $\boldsymbol{z}^{k+1} \leftarrow \arg\min_{\boldsymbol{z}} \left(g(\boldsymbol{z}) + \frac{\rho}{2} \|\mathbf{A}\boldsymbol{\beta}^{k+1} + \mathbf{B}\boldsymbol{z} - \mathbf{c} + \boldsymbol{u}^k\|_2^2 \right)$
    $\boldsymbol{u}^{k+1} \leftarrow \boldsymbol{u}^k + \mathbf{A}\boldsymbol{\beta}^{k+1} + \mathbf{B}\boldsymbol{z}^{k+1} - \mathbf{c}$
    $k \leftarrow k + 1$
  until convergence criteria are met
  return $\boldsymbol{\beta}$, $\boldsymbol{z}$, $\boldsymbol{u}$
end procedure

2.4 Network Sparsification

2.4.1 Naive Sparsifiers

Uniform Sparsifiers

Suppose we want q edges for the sparsified graph. For Uniform sparsifiers, if it is a weighted graph, we sample q edges from $\boldsymbol{E}$ without replacement, with the probability $P(e)$ of selecting an edge e given by

$$P(e) = \frac{w(e)}{\sum_{e' \in \boldsymbol{E}} w(e')}.$$

For unweighted graphs, we sample q edges at random from $\boldsymbol{E}$ without replacement, where each edge e has an equal probability:

$$P(e) = \frac{1}{|\boldsymbol{E}|}.$$

The advantage of this sparsifier is that it is very simple to construct and easy to understand, and it does not create excessive edges. Figure 2.3 is an illustration of a uniform sparsifier.

Figure 2.3: Illustration of Uniform sparsifiers.

Chain Sparsifiers

Chain sparsifiers are defined as a Depth First Search (DFS) on the original graph from a random starting point, also known as Spanning Tree 1. Nodes are connected in the order they are visited by DFS. Note that Chain sparsifiers will create edges that are not in $\boldsymbol{G}$. For instance, if i and j are two successive nodes visited by DFS, there may not be an edge that directly connects i and j in $\boldsymbol{G}$. However, there will be a path that travels from i to j through some other nodes in $\boldsymbol{G}$. Mathematically, suppose we initiate DFS from a random vertex $v \in \boldsymbol{V}$. Let π denote the sequence of vertices DFS visited: $\pi = [v_1, v_2, \ldots, v_n]$ where $v_1 = v$ is the starting vertex. We construct the edge set $\hat{\boldsymbol{E}}$ for the sparsifier $\hat{\boldsymbol{G}}$ by connecting each pair of consecutively visited vertices. Formally,

$$\hat{\boldsymbol{E}} = \{(v_i, v_{i+1}) \mid 1 \leq i < n\}.$$

Figure 2.4 is an illustration of Chain sparsifiers.

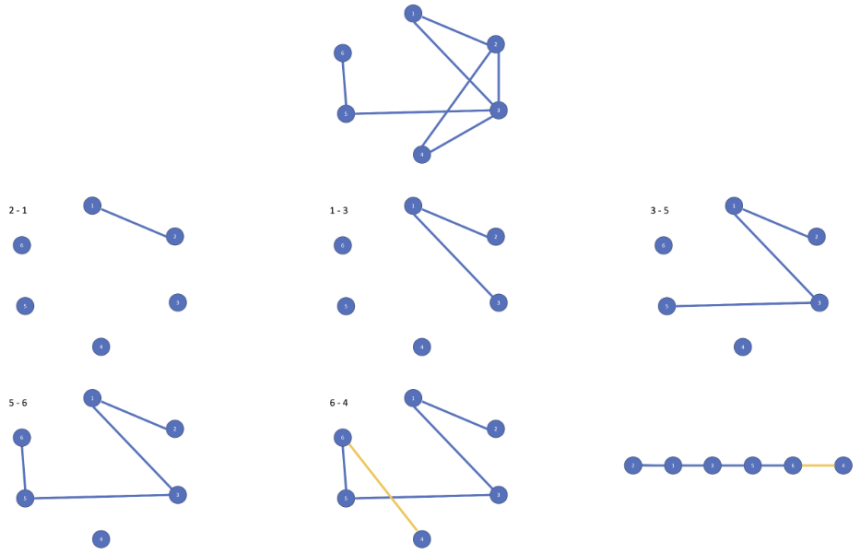


Figure 2.4: Illustration of Chain Sparsifiers.

Spanning Tree Sparsifiers

For Spanning Trees, we make variations on the number of leaves connected to the parents, which is parameterized by n. We begin by randomly selecting a node $u \in \boldsymbol{V}$ to start the sparsification process. Set $\boldsymbol{V'} = \{u\}$ and $\hat{\boldsymbol{E}} = \emptyset$. For the current node u, find all adjacent nodes $\{v \in \boldsymbol{V} \mid (u, v) \in \boldsymbol{E}\}$ that are not yet included in $\boldsymbol{V'}$. Sample n of these nodes to connect to u if there are at least n available. If fewer than n nodes are available, connect u to all of them. Add these nodes to $\boldsymbol{V'}$ and the corresponding edges to $\hat{\boldsymbol{E}}$. Recursively apply this process to each newly added node, treating each as u and repeating the expansion. Mathematically, for each node u in

$\boldsymbol{V'}$ (starting with the initial node), perform:

$$\boldsymbol{A}_u = \{v \in \boldsymbol{V} \setminus \boldsymbol{V'} \mid (u, v) \in \boldsymbol{E}\},$$
$$\boldsymbol{B}_u = \text{Sample } \min(n, |\boldsymbol{A}_u|) \text{ nodes from } \boldsymbol{A}_u,$$
$$\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v) \mid v \in \boldsymbol{B}_u\},$$
$$\boldsymbol{V'} \leftarrow \boldsymbol{V'} \cup \boldsymbol{B}_u.$$

If all nodes in $\boldsymbol{V'}$ are exhausted (all possible connections made), but $\boldsymbol{V'} \neq \boldsymbol{V}$, meaning not all vertices are included, then identify a node $\tilde{u} \in \boldsymbol{V'}$ such that there exists an edge $e = (\tilde{u}, \tilde{s}) \in \boldsymbol{E}$ with $\tilde{s} \notin \boldsymbol{V'}$. Add e to $\hat{\boldsymbol{E}}$ and $\tilde{s}$ to $\boldsymbol{V'}$. Continue the expansion process from $\tilde{s}$. Specifically, the backtracking process goes like:

While $|\boldsymbol{V'}| < |\boldsymbol{V}|$:

Find $\tilde{u} \in \boldsymbol{V'}$ such that $\exists(\tilde{u}, \tilde{s}) \in \boldsymbol{E}$ with $\tilde{s} \notin \boldsymbol{V'}$,

$\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(\tilde{u}, \tilde{s})\}$,

$\boldsymbol{V'} \leftarrow \boldsymbol{V'} \cup \{\tilde{s}\}$,

Continue expansion from $\tilde{s}$.

Figure 2.5 is an illustration of Spanning Tree sparsifiers.

Minimum Spanning Tree

We perform Minimum Spanning Tree (MST) using the `mst` function from R. For a weighted graph, this algorithm searches for a path that connects all nodes with the least total weight. Specifically, let $w : \boldsymbol{E} \to \mathbb{R}^+$ define positive weights for each edge. The goal is to construct a Minimum Spanning Tree $\hat{\boldsymbol{G}} = (\boldsymbol{V}, \hat{\boldsymbol{E}})$ that connects all vertices with the minimum total weight. Start with an empty set of edges $\hat{\boldsymbol{E}} = \emptyset$

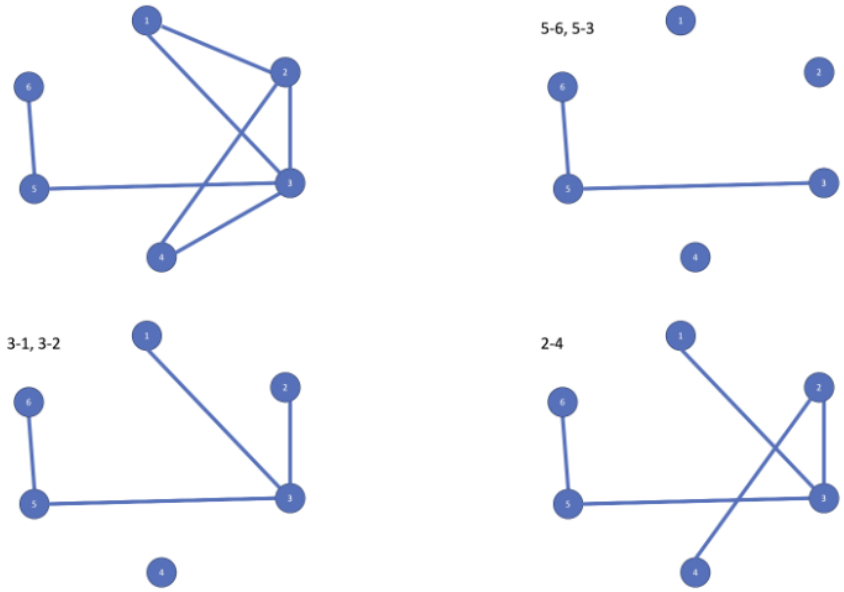


Figure 2.5: Illustration of Spanning Tree Sparsifiers.

and select an arbitrary vertex $v \in \boldsymbol{V}$ to begin the MST. Initialize the MST vertex set $\boldsymbol{V'} = \{v\}$. At each step, select the edge $e = (u, v)$ with the minimum weight $w(e)$ that connects a vertex in the partial MST $\hat{\boldsymbol{G}}$ to a vertex not yet in $\hat{\boldsymbol{G}}$:

$$e = \arg\min\{w(e') \mid e' \in \boldsymbol{E} \text{ and } e' \text{ connects } v \in \boldsymbol{V'} \text{ to } u \in \boldsymbol{V} \setminus \boldsymbol{V'}\}.$$

The process repeats until all vertices are included in $\boldsymbol{V'}$.

For an unweighted graph, the procedure is the same as finding a spanning tree. The easiest way would be applying Breadth First Search (BFS) on the original network. Select an arbitrary starting vertex $v \in \boldsymbol{V}$. Initialize $\hat{\boldsymbol{G}} = (\boldsymbol{V'}, \hat{\boldsymbol{E}})$ where $\boldsymbol{V'} = \{v\}$ and $\hat{\boldsymbol{E}} = \emptyset$. Use a queue Q with v as the initial element. While $Q \neq \emptyset$: Dequeue the first vertex u from Q. Then for each adjacent vertex w of u where

$(u, w) \in \boldsymbol{E}$ and $w \notin \boldsymbol{V'}$:

$$\text{Enqueue } w \text{ into } Q, \quad \boldsymbol{V'} \leftarrow \boldsymbol{V'} \cup \{w\}, \quad \hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, w)\}.$$

The process repeats until all vertices are included in $\boldsymbol{V'}$. Figure 2.6 is an illustration of the Minimum Spanning Tree sparsifier.

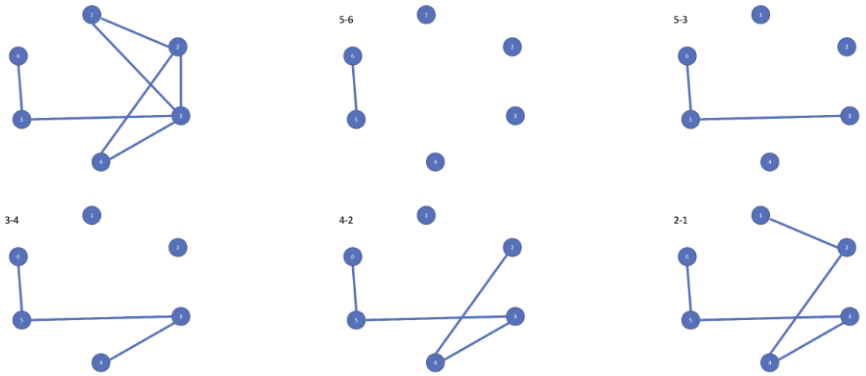


Figure 2.6: Illustration of Minimum Spanning Tree sparsifiers.

Star Sparsifiers

Star sparsification is performed by connecting all the nodes to the one with the highest degrees. First, we identify the top three vertices V_{top} with the highest degrees:

$$V_{\text{top}} = \{\, v \in \boldsymbol{V} : \deg(v) \text{ is among the three highest} \,\}$$

and randomly select one vertex v^* from V_{top}. Construct $\hat{\boldsymbol{G}} = (\boldsymbol{V}, \hat{\boldsymbol{E}})$ with:

$$\hat{\boldsymbol{E}} = \{(v^*, v) : v \in \boldsymbol{V} \setminus \{v^*\}\}.$$

Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a graph with vertices $\boldsymbol{V}$ and edges $\boldsymbol{E}$. The objective is to create a Star sparsifier $\hat{\boldsymbol{G}}$ centered on a node with one of the highest degrees. Figure 2.7 is an illustration of the Star sparsifier.

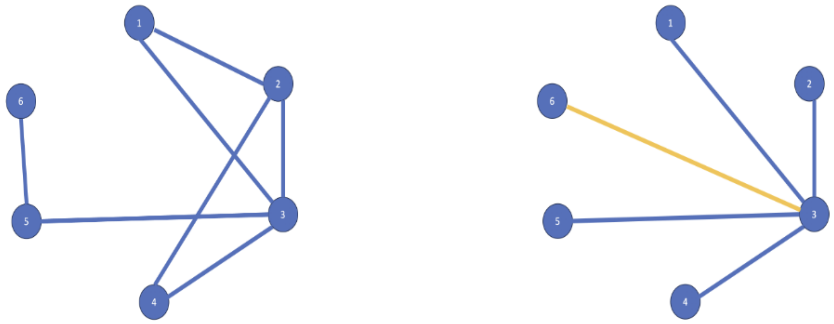


Figure 2.7: Illustration of Star sparsifiers.

2.4.2 Proposed Sparsifiers

KN-based Sparsifiers

For KN-based sparsifiers, the idea is to visit each node exactly once and connect each node with k of its neighbors. For weighted graphs, suppose we have a weight function $w : \boldsymbol{E} \to \mathbb{R}^+$. For each vertex $u \in \boldsymbol{V}$, we calculate $d_u = |\{v \in \boldsymbol{V} : (u, v) \in \boldsymbol{E}\}|$ and $W_u = \sum_{v \in N(u)} w_{u,v}$. If $d_u \leq k$, include all edges (u, v) for $v \in N(u)$ in $\hat{\boldsymbol{E}}$ with weights $\frac{w_{u,v}}{2}$. If $d_u > k$, sample k edges from $N(u)$ with probabilities

$$P_{uv} = \frac{w_{u,v}}{W_u},$$

with replacement. Add these edges to $\hat{\boldsymbol{E}}$ with weight $\frac{W_u}{2k}$ each. Construct $\hat{\boldsymbol{G}} = (\boldsymbol{V}, \hat{\boldsymbol{E}}, \hat{w})$ where $\hat{w}$ is adjusted based on the above criteria. For the unweighted sparsifier, the difference lies in the edge selection step. If $d_u \leq k$, all incident edges

of vertex u are included in the sparsified graph:

$$\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v) : v \in N(u)\}.$$

If $d_u > k$, sample k edges uniformly from the set of all edges incident to u:

$$\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \text{Sample}_k(\{(u, v) : v \in N(u)\}).$$

The advantage of KN is that edges are sampled with probability adjusted to the degree of nodes. However, it takes a relatively long time to implement. An unweighted version of the KN algorithm is shown in Algorithm 4. Figure 2.8 is an illustration of KN sparsifiers.

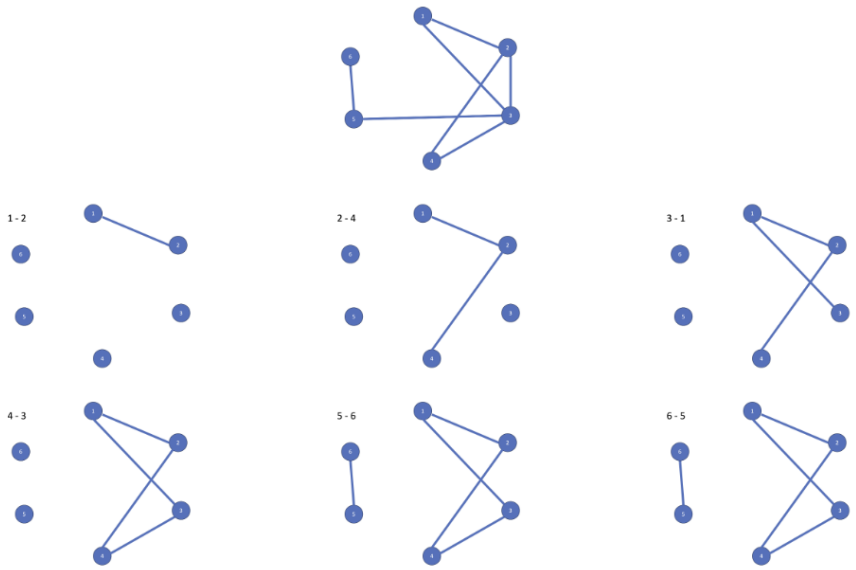


Figure 2.8: Illustration of KN sparsifiers.

Algorithm 4 KN Sparsifiers for Unweighted Graphs

procedure KN-UNWEIGHTED($\boldsymbol{G}, k$)
  Input: Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, integer k
  Output: Sparsified graph $\hat{\boldsymbol{G}} = (\boldsymbol{V}, \hat{\boldsymbol{E}})$
  Initialize $\hat{\boldsymbol{E}} \leftarrow \emptyset$
  for each vertex $u \in \boldsymbol{V}$ **do**
    Calculate $d_u = |\{v \in \boldsymbol{V} : (u, v) \in \boldsymbol{E}\}|$
    if $d_u \leq k$ **then**
      for each $v \in N(u)$ **do**
        $\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v)\}$
      end for
    else
      Sample set $S(u)$ of k edges uniformly from $N(u)$
      for each $(u, v) \in S(u)$ **do**
        $\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v)\}$
      end for
    end if
  end for
end procedure

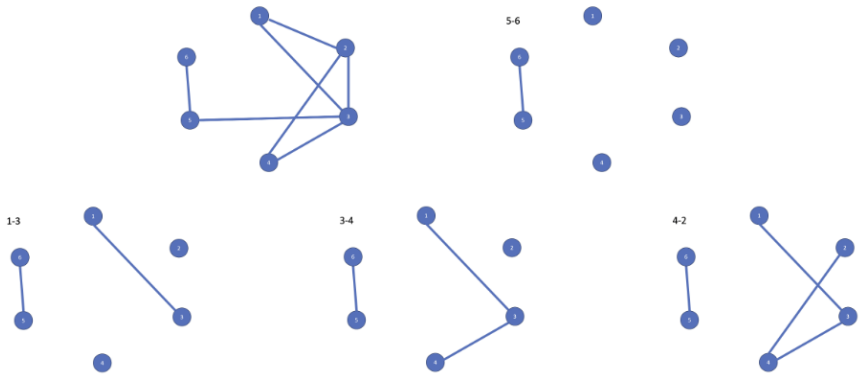


Figure 2.9: Illustration of random walk sparsifiers.

Random Walk Sparsifiers

For random walk sparsifiers, the parameter n defines how many nodes each node connects to. A node u is randomly selected from $\boldsymbol{V}$ to start the sparsification process. For node u, we identify neighbors $N(u)$ not yet in $\hat{\boldsymbol{G}}$ and calculate $d_u = |N(u)|$.

There are two possible cases:

$$\text{If } d_u \leq n, \quad \hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v) : v \in N(u)\},$$

$$\text{and if } d_u > n, \quad \hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v_i) : v_i \text{ is sampled from } N(u)\}.$$

Then we move on to its child nodes and repeat this process. If we go through all the nodes in the sparsifier and $\boldsymbol{V} \neq \hat{\boldsymbol{V}}$, we restart with a new node from $\boldsymbol{V} \setminus \hat{\boldsymbol{V}}$. The detailed algorithm is shown in Algorithm 5. Figure 2.9 is an illustration of the Random Walk sparsifier.

In Figure 2.10, we demonstrate an illustration of a full graph and its corresponding sparsified graphs. The full graph is a DCSBM (Degree-Corrected Stochastic Block Model) graph with two clusters, and we plot the sparsified graph using seven different methods. Notice that the solid black lines represent edges that exist in the original graph, whereas yellow lines represent edges that did not exist in the original graph.

2.4.3 Weighted Sparsifiers

Recall that $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is an undirected graph with vertex set $\boldsymbol{V} = \{1, 2, \ldots, n\}$, and edge set $\boldsymbol{E} \subseteq \{(i, j) : i, j \in \boldsymbol{V}\}$. Define $\tilde{\boldsymbol{G}} = (\boldsymbol{V}, \tilde{\boldsymbol{E}})$ as a sparsified graph of $\boldsymbol{G}$ with $\tilde{\boldsymbol{E}} \subseteq \boldsymbol{E}$.

Given a node i, $\forall k$ such that $(i, j_k) \in \boldsymbol{E}$, $k = 1, 2, \ldots, K$, unweighted sparsifiers will connect two nodes such that

$$P((i, j_k) \in \tilde{\boldsymbol{E}}) = \frac{1}{K} \quad \forall k.$$

Algorithm 5 Random Walk Sparsifiers

procedure RANDOMWALK($\boldsymbol{G}, n$)
    Input: Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, integer n
    Output: Sparsified graph $\hat{\boldsymbol{G}} = (\hat{\boldsymbol{V}}, \hat{\boldsymbol{E}})$
    Initialize $\hat{\boldsymbol{V}} \leftarrow \emptyset$, $\hat{\boldsymbol{E}} \leftarrow \emptyset$
    Choose a random node u from $\boldsymbol{V}$
    Initialize visited $\leftarrow \{u\}$, stack $\leftarrow [u]$
    while stack $\neq \emptyset$ **do**
        $u \leftarrow$ stack.pop()
        Add u to $\hat{\boldsymbol{V}}$
        Identify $N(u) \setminus \hat{\boldsymbol{V}}$ (neighbors not yet in $\hat{\boldsymbol{G}}$)
        Calculate $d_u = |N(u) \setminus \hat{\boldsymbol{V}}|$
        if $d_u \leq n$ **then**
            for each $v \in N(u) \setminus \hat{\boldsymbol{V}}$ **do**
                $\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v)\}$
                if $v \notin$ visited **then**
                    stack.push(v)
                    visited.add(v)
                end if
            end for
        else
            Sample n nodes $\{v_i\}$ uniformly from $N(u) \setminus \hat{\boldsymbol{V}}$
            for each v_i in sampled nodes **do**
                $\hat{\boldsymbol{E}} \leftarrow \hat{\boldsymbol{E}} \cup \{(u, v_i)\}$
                if $v_i \notin$ visited **then**
                    stack.push(v_i)
                    visited.add(v_i)
                end if
            end for
        end if
        if $|\hat{\boldsymbol{V}}| < |\boldsymbol{V}|$ and stack $= \emptyset$ **then**
            Sample a node $\tilde{u}$ from $\boldsymbol{V} \setminus$ visited
            stack.push($\tilde{u}$)
            visited.add($\tilde{u}$)
        end if
    end while
end procedure

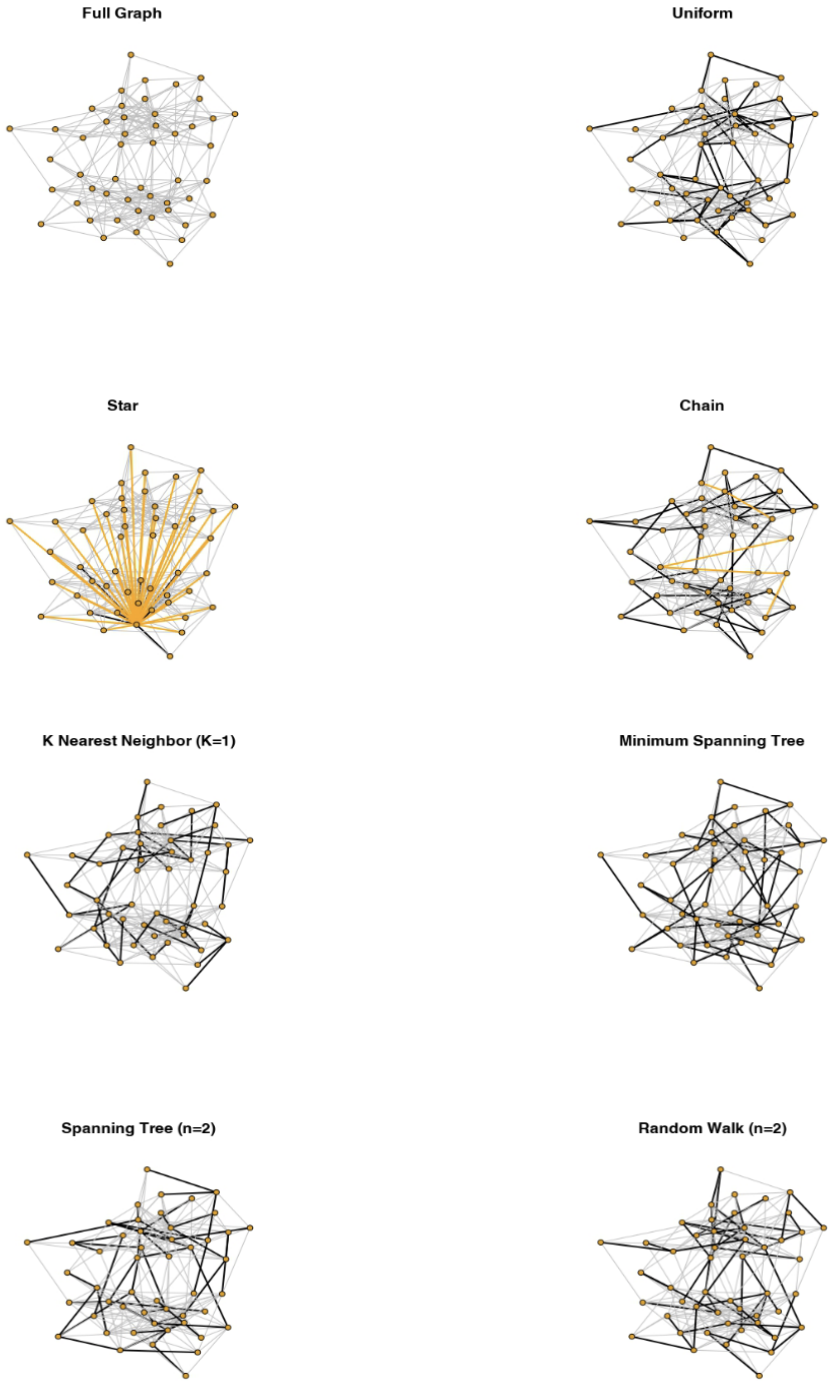


Figure 2.10: Illustration of sparsifiers.

For weighted sparsifiers,

$$P((i, j_k) \in \tilde{\boldsymbol{E}}) \propto \frac{1}{|y_i - y_{j_k}|}.$$

Essentially, we define the weighted sparsifiers in a way that nodes with similar observed values are more likely to be connected.

2.5 Simulations

2.5.1 Evaluation Metric

Define $\boldsymbol{\beta}_{\text{true}}$ as the true underlying signal, $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$ as the estimation using the full graph, and $\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}}$ as the estimation using the sparsified graph.

We have the following metrics to evaluate performances:

- $\|\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{\text{true}}\|_2$: ℓ_2 norm of the difference between the estimation and the true signal. This is also known as the Root Sum of Squares (RSS).

- $\|\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}\|_2$: ℓ_2 norm of the difference between the estimation using the sparsified graph and the solution from the full graph. This measures the RSS with respect to the full graph.

- Sparsity: Number of unique β_i (Degree of Freedom).

- Computation Time: Number of seconds it takes to solve the optimization problem.

2.5.2 Comparison on Different Graphs

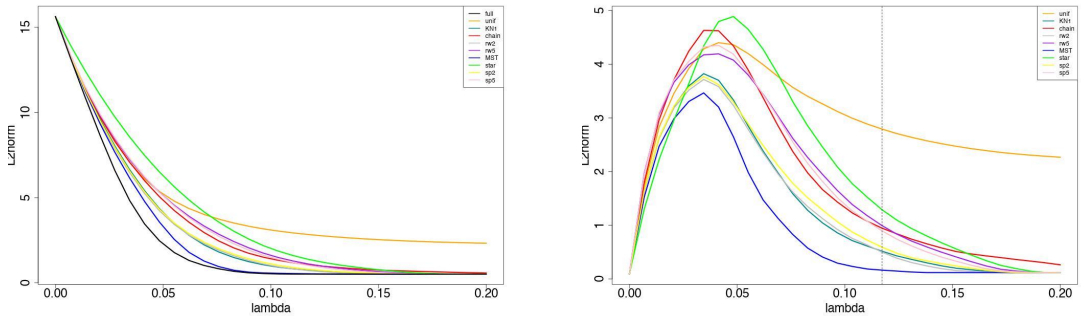
We study the performance of different sparsifiers at a fixed number of edges. Each experiment is implemented over 5 replications, and the plots show the mean of the replications. For all experiments other than the Simple Random graph, we let $C = \{C_1, \ldots, C_{10}\}$ be the set of clusters. We set $\beta_i = k$ for all $i \in C_k$; $y_i = \beta_i + \epsilon_i$, where $\epsilon_i \sim N(0, 0.5)$. For the evaluation metric, we choose to use $\|\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{\text{true}}\|_2$ and $\|\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}\|_2$, which compare the estimated coefficients with both the true signal and the signal from the full graph. Additionally, for sparsification, we choose to use sparsifiers with 2000 edges for all sparsifiers.

Simple Random Graph

For the Simple Random graph, edges are randomly distributed among the nodes. There is only one cluster, which means there is no grouping structure within the network. For the first experiment, we define a simple graph with 1000 nodes and 20000 edges. From Figure 2.11 panel (a), we can see that the overall trend is decreasing for every sparsifier. This means that as λ increases, the estimated coefficients get closer and closer to the true value. In panel (b), the dotted line represents the λ where the estimation from the full graph is the closest to the true coefficient. Among these sparsifiers, the Minimum Spanning Tree has the best performance among all of them, both when comparing with $\boldsymbol{\beta}_{\text{true}}$ and $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$. The Uniform sparsifier has the worst estimation.

Degree Corrected Stochastic Block Model (DCSBM) Graph

The Degree Corrected Stochastic Block Model (DCSBM) is an extension of the Stochastic Block Model, a common model for network data. The stochastic block



(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{G}||_2$ vs. λ

Figure 2.11: RSS based on Simple Random graph.

model partitions nodes into blocks. Within each block, the likelihood of connecting any two nodes is identical. This model is widely used in various domains such as social networks, biological networks, and information networks to help understand community structures. However, the biggest problem of the Stochastic Block Model is the assumption of homogeneity within blocks. In each community, the distribution of node degrees is almost uniform, which is not representative of real-world scenarios. A simple example is a social network, where influencers have far more connections than the rest of the people, even if they belong to the same social circle.

To overcome this limitation, a degree-correction term is introduced to allow for variability in node degrees within the same community. Specifically, each node has an expected degree to form connections, which allows the Degree-Corrected Stochastic Block Model to model degree heterogeneity within communities.

For a graph $\boldsymbol{G}$, the nodes are partitioned into K blocks, and each node i belongs to a block denoted by g_i. Each node i is also assigned a degree parameter θ_i, which influences its degree. The probability of an edge between nodes i and j depends on their block memberships (g_i and g_j) and their degree parameters (θ_i and θ_j),

typically modeled as

$$P(i,j) = \theta_i \theta_j \eta_{g_i,g_j},$$

where η_{g_i,g_j} is a parameter that depends on the communities of the nodes and represents the baseline affinity between blocks.

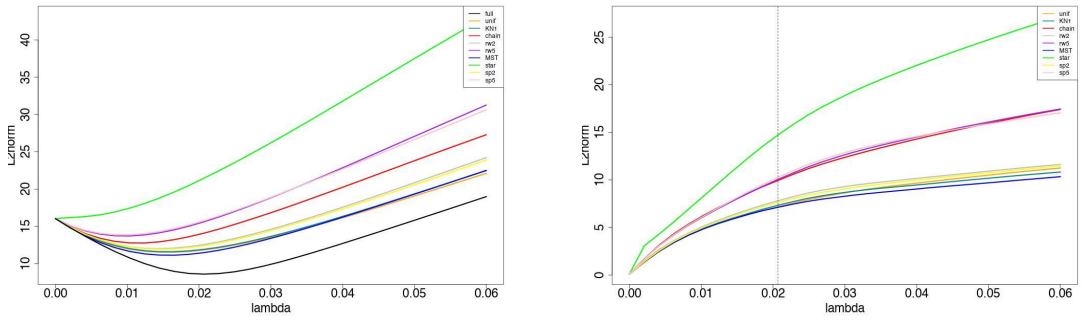
There are a few parameters for this model. The variable β controls the out-in ratio: the ratio of between-block edges over within-block edges. The parameter ρ is the proportion of small degrees within each community if the degrees are from a two-point mass distribution. $\rho > 0$ gives a degree corrected block model.

For this experiment, we use a DCSBM graph with 10 clusters. The graph has 1000 nodes and roughly 50000 edges. The proportion of small degrees within each community is 0.1. The ratio of between-block edges over within-block edges is fixed at 0.1.

From Figure 2.12, Uniform, KN and MST have similar estimation with respect to $\boldsymbol{\beta}_{\text{true}}$ and are more accurate than other sparsifiers. When comparing with $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$, these three sparsifiers have very similar performances and are better than the rest. The Star sparsifier has the worst performance. This is reasonable since the Star is a very centralized structure and it ignores the grouping structure in the original graph.

Cluster Graph

We construct a Cluster graph with 1000 nodes and 50000 edges. There are three variables in this graph: cluster size, the number of edges within each cluster, and the number of edges between each pair of clusters. Within each cluster, all the edges are randomly connected. For this experiment, each cluster has a random cluster size that ranges between 50 and 250, and the density within each cluster also varies.

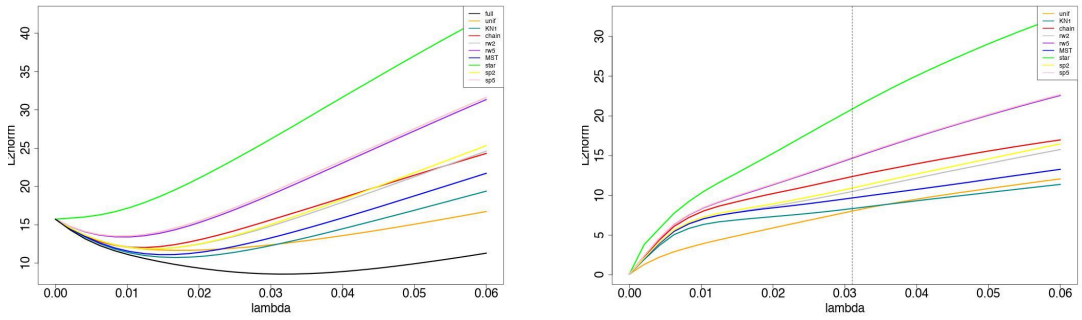


(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ

Figure 2.12: RSS based on DCSBM graph.

The graph is fully connected, with the lowest degree being 7 and the highest degree being 236. The difference between the Cluster graph and DCSBM is that we can control the cluster size and density for each cluster. Also, within each cluster, the distribution of node degree is almost uniform.

From Figure 2.13, Uniform, MST, and KN have the best estimation with respect to $\boldsymbol{\beta}_{\text{true}}$. When comparing with $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$, Uniform and KN are the optimal ones. It is also worth noting that the Star sparsifier has the worst estimation in both cases, since it totally disregards the original grouping structure within the network data. This result is very similar to the conclusion we derived from the last experiment.



(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ

Figure 2.13: RSS based on Cluster graph.

Preferential Attachment (PA) Graph

The Preferential Attachment Model is a foundational model in network theory. Central to this model are two parameters: m_0, representing the initial number of nodes in the network, and m, indicating the number of connections a new node forms when entering into the network. The parameter m_0 establishes the initial network structure, and m governs how fast the network expands by determining the connectivity degree each new node achieves with existing nodes.

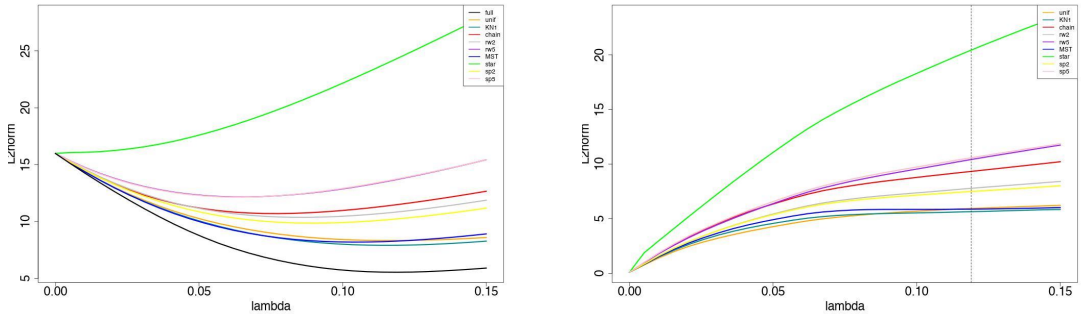
As the network grows, each new node preferentially attaches to existing nodes with a higher degree of connectivity with a probability $\Pi(k_i) = \frac{k_i}{\sum_j k_j}$, where k_i is the degree of a particular node i, and the summation is over all nodes currently in the network. This ensures that nodes with greater connectivity are more likely to receive new links.

The degree distribution adheres to a power law, $P(k) \sim k^{-\gamma}$, where $P(k)$ denotes the probability of randomly selecting a node with degree k, and γ is a constant typically within the range of 2 to 3 for real-world networks. Because of this distribution, the network always has a few highly connected hubs among a vast majority of low-degree nodes, resulting in a heterogeneous network topology. Due to its robustness against random failures and susceptibility to targeted attacks on significant nodes, the Preferential Attachment model serves as a pivotal tool in the examination and construction of networks across various fields, from technology to biology.

For this experiment, we have a PA graph with 10 clusters. The graph has 1000 nodes and approximately 30000 edges. Each cluster has 100 nodes, with power being 3 within each cluster. This means that each cluster has a very similar structure, with the same degree distribution among all the nodes.

From Figure 2.14, again, Uniform, MST and KN have the best estimation with

respect to β_{true} and $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$. The three of them have very similar estimation, and the Star sparsifier is still the worst among all of them. This result is consistent with the previous setups, as they all have a clear grouping structure embedded in the network.



(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ

Figure 2.14: RSS based on PA graph.

Sparsity

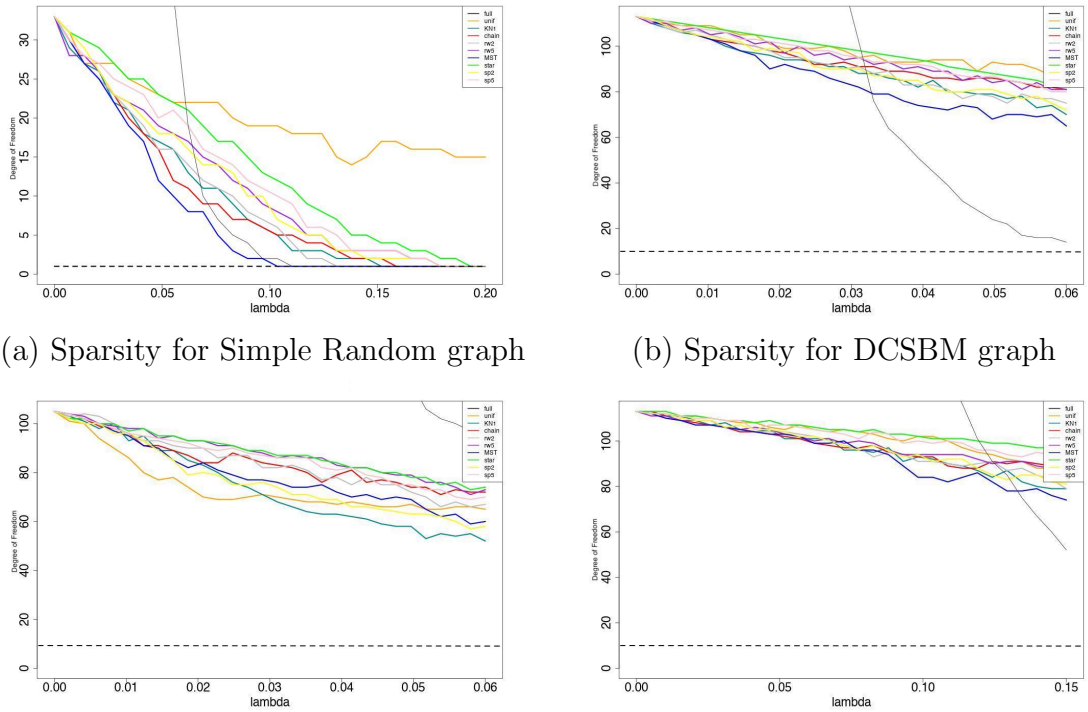
In a Fused LASSO problem, sparsity of the solution is defined by the number of unique coefficients in $\hat{\boldsymbol{\beta}}$. Theoretically, for a fully connected graph, when λ is large enough, the number of unique coefficients will become 1, as all the values converge to the mean of $\boldsymbol{y}$.

The sparsity of the solution depends on the sparsifiers, the number of edges, as well as the centrality of graphs. Usually, if the sparsified graph is not fully connected, some nodes will not be penalized due to the nature of the penalty term. By increasing λ, the coefficients of those isolated nodes will converge to the mean of the subsetted $\boldsymbol{y}$.

We plot the sparsity of each graph by counting the number of unique $\hat{\boldsymbol{\beta}}$ elements at each λ value. The dotted line indicates the correct degree of freedom for each

setup. Theoretically speaking, when λ is large enough, as long as the sparsified graph is fully connected, it can always converge to the true degree of freedom.

From Figure 2.15, all the sparsifiers except Uniform can shrink to the true degree of freedom for the Simple graph. For all other setups, MST and KN can always achieve better sparsity compared to other sparsifiers. For the Cluster graph, the sparsities of sparsifiers are even better than that of the full graph at the same λ value.



(a) Sparsity for Simple Random graph

(b) Sparsity for DCSBM graph

(c) Sparsity for Cluster graph

(d) Sparsity for PA graph

Figure 2.15: Sparsity of four different types of graphs.

2.5.3 Sparsifiers with Different Density

By increasing the number of edges, the likelihood of obtaining a fully connected graph also increases. For instance, consider a fully random graph with 1000 nodes

and 50000 edges. If we randomly select 1000 edges from the graph, on average only 860 unique nodes will be selected. Therefore, it is nearly impossible to get a fully connected graph. If we increase the number of edges to 3000, then the probability increases to 8%. By further increasing the number of edges to 5000, the probability becomes over 95%.

In general, the number of edges in the sparsifier is directly related to the RSS (Root Sum of Squares). In our simulation studies, we choose to experiment with 1000, 2000, and 3000 edges and compute the corresponding average RSS over a small range of λ.

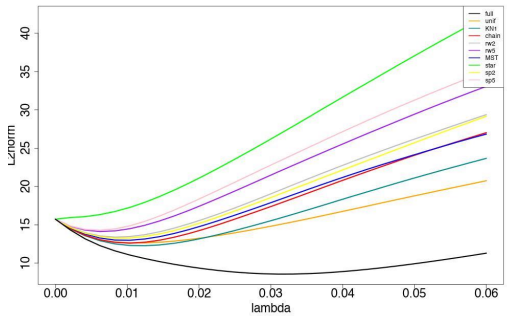
Cluster Graph

For the first experiment, we present the average RSS obtained by performing sparsification on a Cluster graph. As shown in Figure 2.16, RSS is inversely related to the number of edges for most of the sparsifiers. However, the decrement in RSS from 2000 edges to 3000 edges is not as significant. When compared with $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$, the order of the sparsifiers remains unchanged.

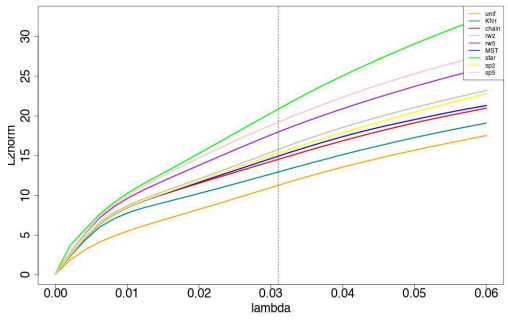
Among all the sparsifiers, Uniform and KN are consistently the most accurate ones. With 3000 edges, the estimations are nearly as accurate as those of the full graph.

DCSBM Graph

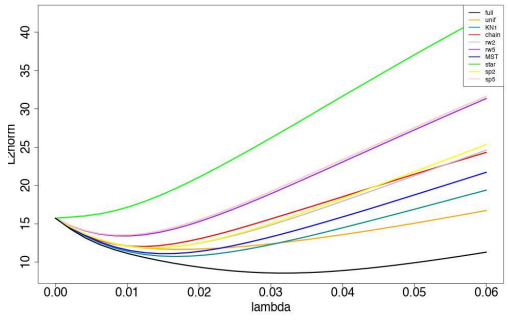
We conduct the same experiment on a DCSBM graph with 10 clusters, with a fixed between-ratio of 0.05. As illustrated in Figure 2.17, the trend closely resembles the previous setup. With 1000 edges, the performances of different sparsifiers are quite similar. However, as the number of edges increases, the differences become more pronounced.



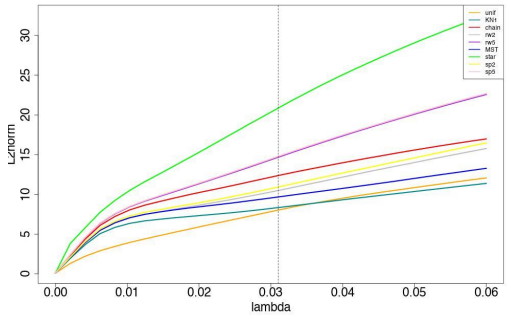
(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (1k edges)



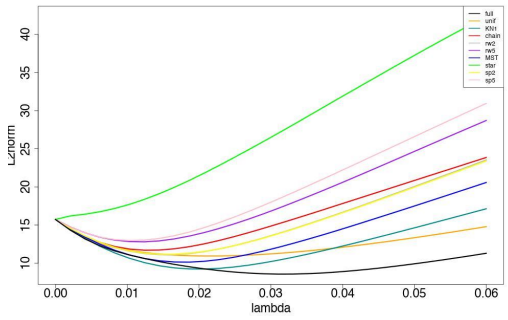
(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (1k edges)



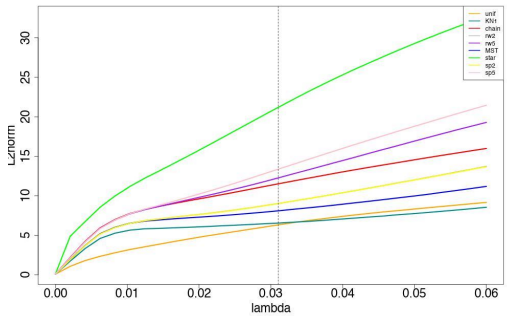
(c) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (2k edges)



(d) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (2k edges)



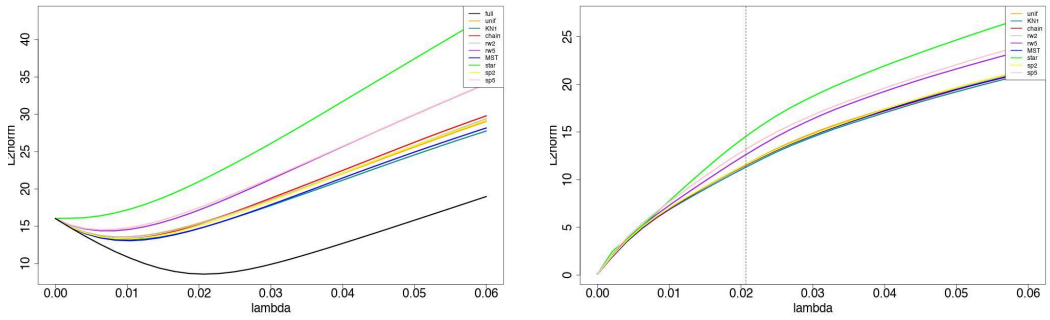
(e) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (3k edges)



(f) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (3k edges)

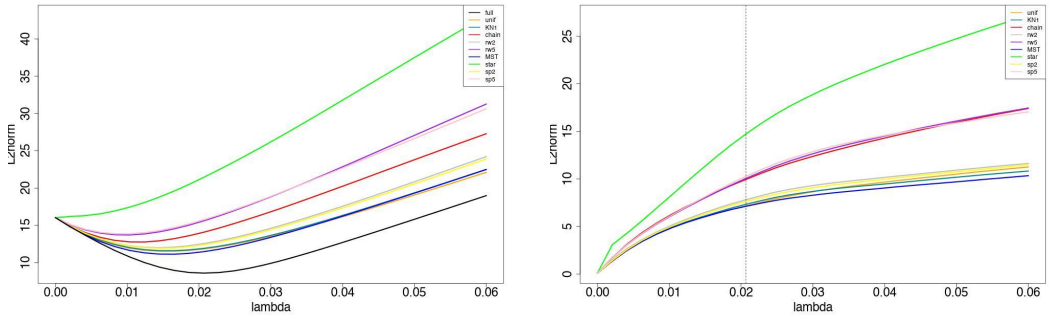
Figure 2.16: RSS for Cluster graph with different number of edges.

Notably, Uniform, KN, Spanning Tree 2 and Random Walk 2 exhibit similar results with 2000 or 3000 edges, and they outperform the other sparsifiers by a significant margin.



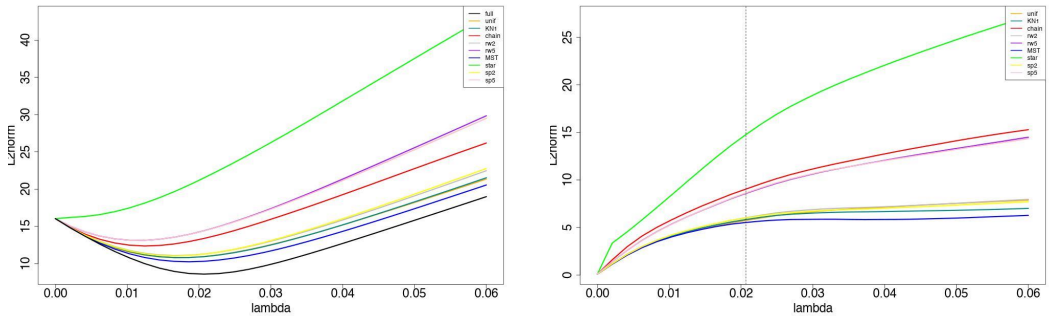
(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (1k edges)

(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (1k edges)



(c) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (2k edges)

(d) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (2k edges)



(e) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (3k edges)

(f) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (3k edges)

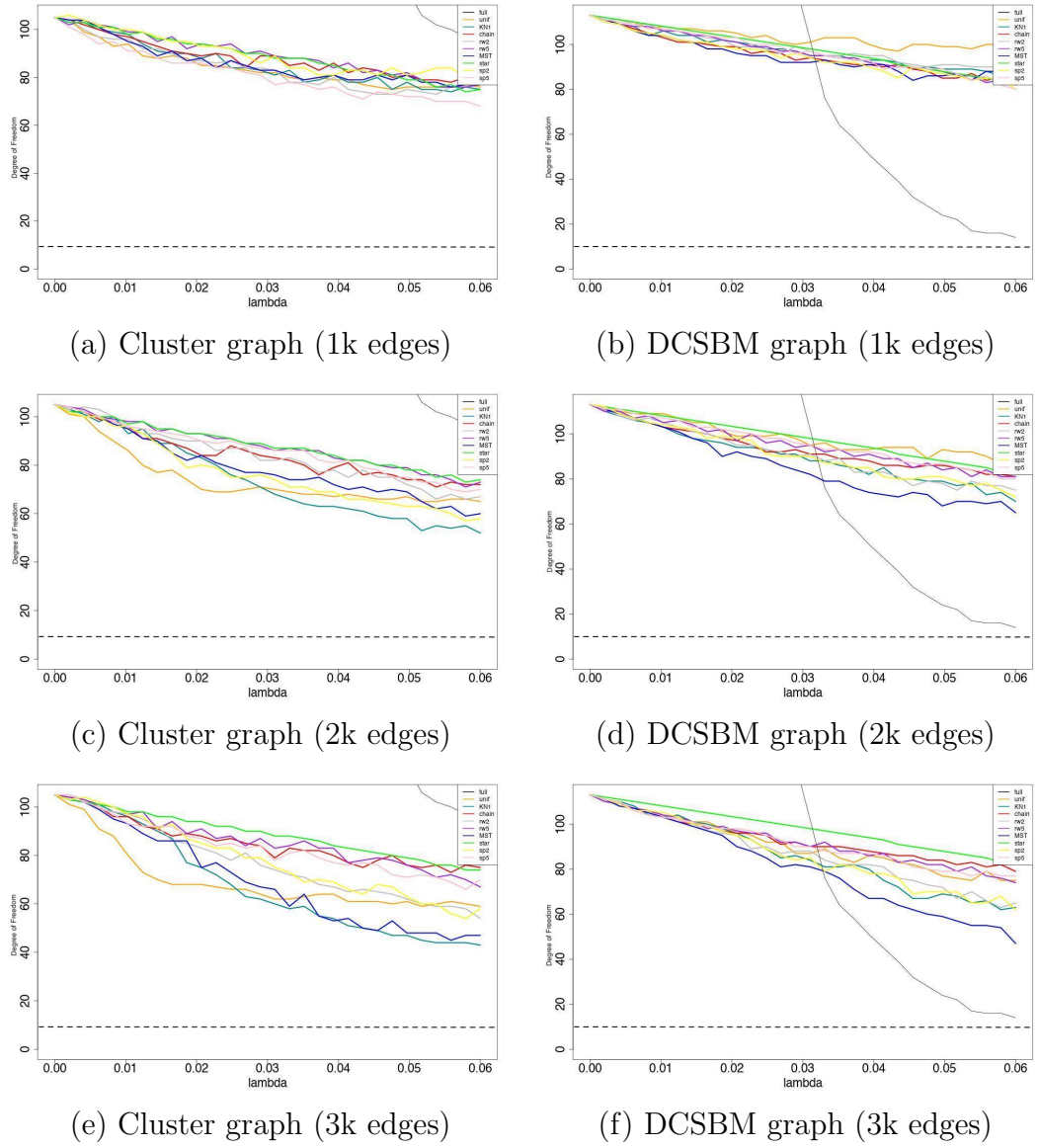
Figure 2.17: RSS for DCSBM graph with different number of edges.

Sparsity

We present the sparsity for each graph at each sparsification level. As depicted in Figure 2.18, as the number of edges increases, the degree of freedom generally decreases. Similar to the estimation results, when the number of edges is 1000, the

difference in sparsity is not significant.

For the Cluster graph, KN exhibits the best sparsity at 2000 edges, and both KN and MST show the best sparsity when the number of edges increases to 3000. Conversely, for the DCSBM graph, MST demonstrates the best sparsity starting from 2000 edges.



(a) Cluster graph (1k edges)

(b) DCSBM graph (1k edges)

(c) Cluster graph (2k edges)

(d) DCSBM graph (2k edges)

(e) Cluster graph (3k edges)

(f) DCSBM graph (3k edges)

Figure 2.18: Sparsity for different sparsifier density.

2.5.4 Centrality of Graphs

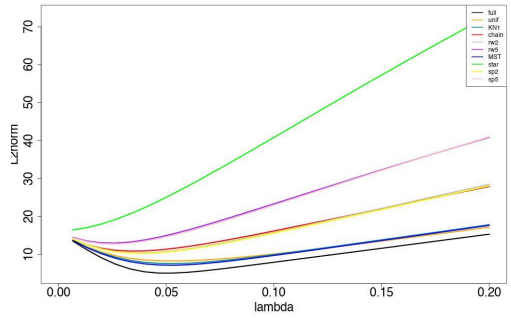
In general, the more centralized the graph is, the less sparse the solutions will be at a particular λ and edge sparsity level. When the graph is purely uniform, meaning every node has an equal chance of connecting to every other node, there is a higher chance that the graph is fully connected at some particular number of edges. This is because no nodes are favored over others, and every node is equally likely to be selected in the sparsifiers. As centrality of the graph increases, some nodes will be selected more likely than others.

We compare the estimation and sparsity for graphs with different centralities in the cluster graph. For low centrality, all the edges in each cluster are connected in a purely random fashion. For high centrality, we randomly select 20 landmarks in each cluster and connect each other node to the landmarks only.

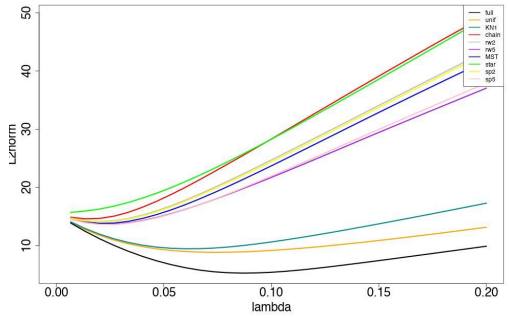
Figure 2.19 and Figure 2.20 show the estimation results and sparsity for two different centralities, respectively. In particular, Uniform and KN stand out when centrality increases. Their estimations become significantly better than the rest when centrality increases. Regarding sparsity, as centrality increases, the degree of freedom is higher at the same penalty level. When centrality is low, MST and KN have the lowest degree of freedom. When centrality becomes higher, KN has the best sparsity.

2.5.5 Weighted vs. Unweighted Sparsifiers

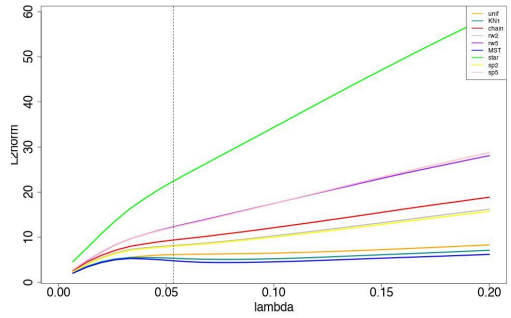
We present the estimation results on a DCSBM graph with 10 clusters. From Figure 2.21, MST and KN have significantly better estimations when going from unweighted to weighted sparsifiers. Their estimations are almost as good as that of the full graph. When comparing with $\hat{\boldsymbol{\beta}}_{\boldsymbol{G}}$, the Uniform sparsifier has the best estimation. This is



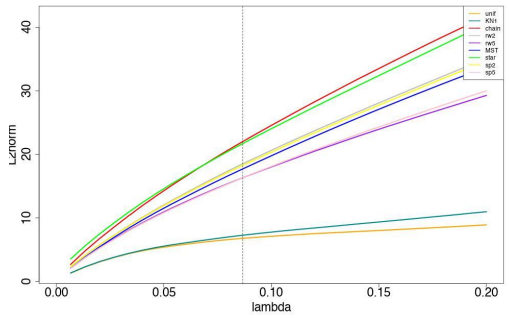
(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (Low Centrality)



(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (High Centrality)

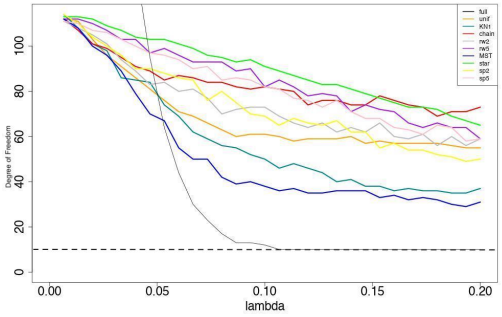


(c) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (Low Centrality)

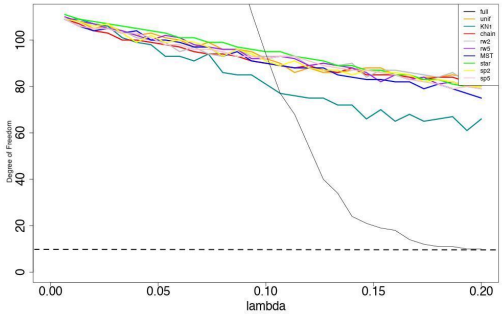


(d) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (High Centrality)

Figure 2.19: RSS for different centrality.



(a) Sparsity for low centrality

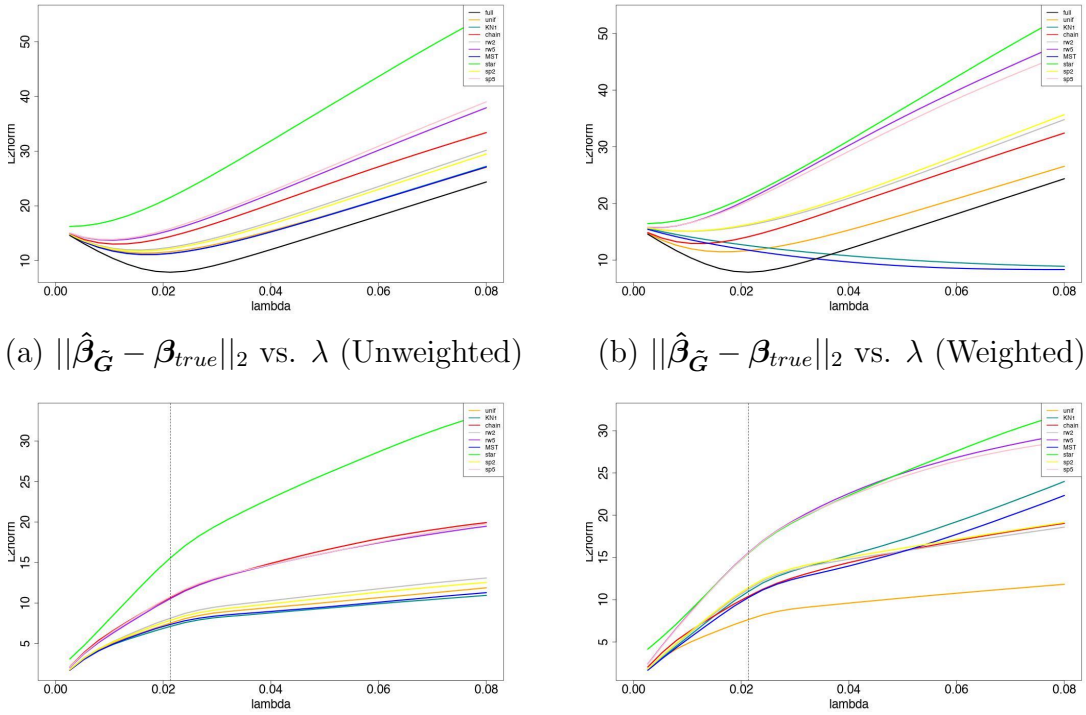


(b) Sparsity for high centrality

Figure 2.20: Sparsity for different centrality.

reasonable since weighted MST and KN show very different trends when comparing to the curve from the full graph. Regarding sparsity, from Figure 2.22, weighted KN and weighted MST are the only algorithms with significantly improved degree

of freedom. In the unweighted case, their sparsities are better than the rest of the sparsifiers, but the difference is not as big. After adding weights to the sparsifiers, the tree methods show no significant change in degree of freedom, but KN and MST benefit a lot from the weights.



(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (Unweighted)

(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (Weighted)

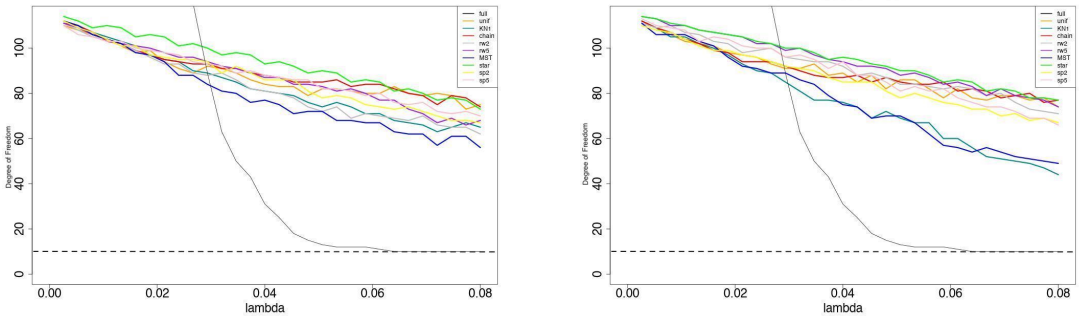
(c) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (Unweighted)

(d) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (Weighted)

Figure 2.21: RSS for weighted and unweighted sparsifiers.

2.5.6 Comparison with Community Detection

We compare our sparsifiers with some common Community Detection algorithms on a DCSBM graph. We show that sparsifiers can solve for coefficients in one step, whereas Community Detection algorithms need to find communities first and then compute averages over each cluster as estimated coefficients. At the same time, when the noise level is high or there are a large number of clusters, sparsifiers can



(a) Sparsity for unweighted sparsifiers (b) Sparsity for weighted sparsifiers

Figure 2.22: Sparsity for weighted and unweighted sparsifiers.

outperform Community Detection algorithms significantly.

Table 2.1 and Table 2.2 show the estimation results for DCSBM graphs with 10 and 20 clusters, respectively. Each column represents a between/within ratio. The larger the ratio, the noisier the graph is. From Table 2.1, we can see that the Community Detection algorithm can capture the signal almost perfectly when the noise level is at 0.1. Sparsifiers show their advantages when we increase it to 0.3. When there are 20 clusters, weighted MST is the third best when the noise level is at 0.1. However, as we increase the between/within ratio to 0.2, all of the top three algorithms are sparsifiers. From this experiment, we can see that sparsifiers perform better than Community Detection algorithms when the number of clusters is large and the noise level is high.

There are several interesting findings from the simulation section. Firstly, weighted KN, MST, and Uniform consistently outperform other sparsifiers in many cases. However, different sparsifiers exhibit varying performances under different graph structures. There is no sparsifier that is universally better than the others. Additionally, depending on the graph structure, results from sparsifiers could be superior to those from the full graph, as sparsifiers can help eliminate noise from the original

Methods	**Between/Within Ratios**		
	0.1	0.2	0.4
MST	33.95	33.9	40.41
KN	35.63	33.48	41.39
Unif	31.29	31.24	**36.9**
MST(w)	11.28	16.72	**21.87**
KN(w)	11.84	16.21	**20.1**
Louvein	9.81	**9.77**	86.97
Walktrap	**0.84**	**2.09**	82.56
Infomap	**0.84**	88.81	90.25
BHMC+SSP	**0.84**	**2.09**	90.26
HCD	8	13.67	89.95

Table 2.1: RSS with respect to true signal for different between/within ratios on a DCSBM graph with 10 clusters.

Methods	**Between/Within Ratios**		
	0.1	0.2	0.4
MST	55.49	67.86	72.65
KN	55.96	**63.4**	**66.3**
Unif	56.04	67.14	70.88
MST(w)	**13.53**	**20.19**	**25.05**
KN(w)	13.96	**16.96**	**20.86**
Louvein	77.84	92.1	183.14
Walktrap	**2.82**	80.78	183.8
Infomap	185.54	180.37	185.51
BHMC+SSP	**6.64**	117.36	185.46
HCD	54	80.35	185.51

Table 2.2: RSS with respect to true signal for different between/within ratios on a DCSBM graph with 20 clusters.

graph structure. Centrality and the number of edges are strongly related to sparsifiers' performances. Generally, as centrality increases, KN and Uniform perform significantly better than the rest. Also, as the number of edges increases, sparsifiers can provide more accurate estimation results. Furthermore, weighted sparsifiers significantly outperform unweighted ones, as they leverage observed values of each node in constructing the sparsifiers. It is also demonstrated that sparsifiers perform better than Community Detection algorithms, especially when the number of clusters is large. Lastly, solving fusion problems with sparsification methods is significantly faster computationally.

2.6 Theoretical Results

In this section, we propose some theoretical results to better understand the estimator's efficiency and accuracy. For the first theorem, we provide an expression for the average squared error of the generalized LASSO estimate $\hat{\boldsymbol{\beta}}$ in reconstructing or estimating the true signal $\boldsymbol{\beta_0}$, adapted from Wang et al. (2016a). If we let $\boldsymbol{\nabla}$ denote the incidence matrix defined in 2.3. Let r denote the number of rows for $\boldsymbol{\nabla}$ and $\boldsymbol{\nabla}^\dagger$ denote the pseudo-inverse of $\boldsymbol{\nabla}$, then we can prove the following result:

Theorem 2.6.1. *Let M denote the maximum ℓ_2 norm of the columns of $\boldsymbol{\nabla}^\dagger$. Then for a tuning parameter value $\lambda = \Theta(M\sqrt{\log r})$, the generalized LASSO estimate $\hat{\boldsymbol{\beta}}$ has average squared error*

$$\frac{\|\hat{\boldsymbol{\beta}} - \boldsymbol{\beta_0}\|_2^2}{n} = O_P\left(\frac{\mathit{nullity}(\boldsymbol{\nabla})}{n} + \frac{M\sqrt{\log r}}{n}\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1\right).$$

Proof. Define $\boldsymbol{y} = \boldsymbol{\beta_0} + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2\boldsymbol{I})$. Define $R = \text{row}(\boldsymbol{\nabla})$, which is the row space of $\boldsymbol{\nabla}$. Define $R^\perp = \text{nullity}(\boldsymbol{\nabla})$. Also, we define $\boldsymbol{P_R}$ as the projection onto

R and $\boldsymbol{P}_{\boldsymbol{R}^\perp}$ as the projection onto $R^\perp$. Since

$$\boldsymbol{y} = (\boldsymbol{P}_{\boldsymbol{R}} + \boldsymbol{P}_{\boldsymbol{R}^\perp})\boldsymbol{y} = \boldsymbol{P}_{\boldsymbol{R}}\boldsymbol{y} + \boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{y},$$

the original estimation problem

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}\in\mathcal{R}^n} \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\nabla}\boldsymbol{\beta}\|_1$$

can be rewritten as

$$\hat{\boldsymbol{\beta}} = \boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{y} + \tilde{\boldsymbol{\beta}},$$

where

$$\tilde{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}\in\mathcal{R}^n} \frac{1}{2}\|\boldsymbol{P}_{\boldsymbol{R}}\boldsymbol{y} - \boldsymbol{\beta}\|_2^2 + \lambda\|\boldsymbol{\nabla}\boldsymbol{\beta}\|_1.$$

Now, denote $\|\boldsymbol{x}\|_R = \|\boldsymbol{P}_{\boldsymbol{R}}\boldsymbol{x}\|_2$ and $\|\boldsymbol{x}\|_{R^\perp} = \|\boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{x}\|_2$, then

$$\begin{aligned}
\|\hat{\boldsymbol{\beta}} - \boldsymbol{\beta_0}\|_2^2 &= \|\boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{y} + \tilde{\boldsymbol{\beta}} - \boldsymbol{\beta_0}\|_2^2 \\
&= \|\boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{\beta_0} + \boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{\epsilon} + \tilde{\boldsymbol{\beta}} - \boldsymbol{\beta_0}\|_2^2 \\
&= \|\boldsymbol{P}_{\boldsymbol{R}^\perp}\boldsymbol{\epsilon} + \tilde{\boldsymbol{\beta}} - \boldsymbol{P}_{\boldsymbol{R}}\boldsymbol{\beta_0}\|_2^2 \\
&= \|\boldsymbol{\epsilon}\|_{R^\perp}^2 + \|\tilde{\boldsymbol{\beta}} - \boldsymbol{\beta_0}\|_R^2.
\end{aligned}$$

By optimality,

$$\begin{aligned}
\frac{1}{2}\|\boldsymbol{y}-\tilde{\boldsymbol{\beta}}\|_R^2+\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 &\leq \frac{1}{2}\|\boldsymbol{y}-\boldsymbol{\beta_0}\|_R^2+\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1 \\
\|\boldsymbol{y}-\tilde{\boldsymbol{\beta}}\|_R^2 &\leq \|\boldsymbol{y}-\boldsymbol{\beta_0}\|_R^2+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 \\
\|\boldsymbol{\beta_0}-\tilde{\boldsymbol{\beta}}+\boldsymbol{\epsilon}\|_R^2 &\leq \|\boldsymbol{\epsilon}\|_R^2+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 \\
\|\boldsymbol{\beta_0}-\tilde{\boldsymbol{\beta}}\|_R^2+2\boldsymbol{\epsilon}^T\boldsymbol{P_R}(\boldsymbol{\beta_0}-\tilde{\boldsymbol{\beta}})+\|\boldsymbol{\epsilon}\|_R^2 &\leq \|\boldsymbol{\epsilon}\|_R^2+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 \\
\|\boldsymbol{\beta_0}-\tilde{\boldsymbol{\beta}}\|_R^2 &\leq 2\boldsymbol{\epsilon}^T\boldsymbol{P_R}(\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0})+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1.
\end{aligned}$$

By definition of projection matrix, $\boldsymbol{P_R}=\boldsymbol{\nabla}^\dagger\boldsymbol{\nabla}$. Also, by Hölder's inequality, for $p,q\in[1,\infty)$, $\frac{1}{p}+\frac{1}{q}=1$,

$$\sum_{k=1}^{n}|a_kb_k|\leq(\sum_{k=1}^{n}|a_k|^p)^{\frac{1}{p}}(\sum_{k=1}^{n}|b_k|^q)^{\frac{1}{q}}.$$

Therefore,

$$\boldsymbol{\epsilon}^T\boldsymbol{\nabla}^\dagger\boldsymbol{\nabla}(\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0})\leq\|(\boldsymbol{\nabla}^\dagger)^{\boldsymbol{T}}\boldsymbol{\epsilon}\|_\infty\|\boldsymbol{\nabla}(\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0})\|_1.$$

If $\lambda\geq\|(\boldsymbol{\nabla}^\dagger)^{\boldsymbol{T}}\boldsymbol{\epsilon}\|_\infty$,

$$\begin{aligned}
\|\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0}\|_R^2 &\leq 2\|(\boldsymbol{\nabla}^\dagger)^{\boldsymbol{T}}\boldsymbol{\epsilon}\|_\infty\|\boldsymbol{\nabla}(\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0})\|_1+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 \\
&\leq 2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}-\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1+2\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1-2\lambda\|\boldsymbol{\nabla}\tilde{\boldsymbol{\beta}}\|_1 \\
&\leq 4\lambda\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1.
\end{aligned}$$

By a standard result on the maximum of Gaussian,

$$\|(\boldsymbol{\nabla}^\dagger)^{\boldsymbol{T}}\boldsymbol{\epsilon}\|_\infty=O_p(M\sqrt{\log r}),$$

$$\|\tilde{\boldsymbol{\beta}}-\boldsymbol{\beta_0}\|_R^2=O_p(M\sqrt{\log r}\|\boldsymbol{\nabla}\boldsymbol{\beta_0}\|_1).$$

□

We move on to prove some results on M, which is the maximum ℓ_2 norm of columns of $\boldsymbol{\nabla}^{\dagger}$. We aim to show that for different graph structures, we will have different M values. The following two lemmas proves the value of M for Star and Chain sparsifiers.

Lemma 2.6.2. *For a star sparsifier with n nodes and $n-1$ edges, $M \to 1$ in probability when $n \to \infty$.*

Proof. Without loss of generality, assume the graph has n nodes and $n-1$ edges and the star sparsifier is centered around node 1. By definition, $\boldsymbol{\nabla_s} \in R^{(n-1)\times n}$ and

$$\boldsymbol{\nabla_s} = \begin{bmatrix} 1 & -1 & 0 & 0 & ... & 0 \\ 1 & 0 & -1 & 0 & ... & 0 \\ 1 & 0 & 0 & -1 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 1 & 0 & 0 & 0 & ... & -1 \end{bmatrix}.$$

Since $\boldsymbol{\nabla_s}$ is full rank, then by definition of pseudo inverse,

$$\boldsymbol{\nabla_s^{\dagger}} = \boldsymbol{\nabla_s^T}(\boldsymbol{\nabla_s}\boldsymbol{\nabla_s^T})^{-1}.$$

Therefore,

$$\boldsymbol{\nabla_s \nabla_s^T} = \begin{bmatrix} 2 & 1 & 1 & 1 & ... & 1 \\ 1 & 2 & 1 & 1 & ... & 1 \\ 1 & 1 & 2 & 1 & ... & 1 \\ ... & ... & ... & ... & ... & ... \\ 1 & 1 & 1 & 1 & ... & 2 \end{bmatrix},$$

$$(\boldsymbol{\nabla_s \nabla_s^T})^{-1} = \begin{bmatrix} \frac{n-1}{n} & -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ -\frac{1}{n} & \frac{n-1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ -\frac{1}{n} & -\frac{1}{n} & \frac{n-1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ ... & ... & ... & ... & ... & ... \\ -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & \frac{n-1}{n} \end{bmatrix},$$

$$\begin{aligned} \boldsymbol{\nabla_s^\dagger} &= \boldsymbol{\nabla_s^T}(\boldsymbol{\nabla_s \nabla_s^T})^{-1} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 & ... & 1 \\ -1 & 0 & 0 & 0 & ... & 0 \\ 0 & -1 & 0 & 0 & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & ... & -1 \end{bmatrix} \begin{bmatrix} \frac{n-1}{n} & -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ -\frac{1}{n} & \frac{n-1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ -\frac{1}{n} & -\frac{1}{n} & \frac{n-1}{n} & -\frac{1}{n} & ... & -\frac{1}{n} \\ ... & ... & ... & ... & ... & ... \\ -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & -\frac{1}{n} & ... & \frac{n-1}{n} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{n-1}{n} & \frac{1}{n} & \frac{1}{n} & \frac{1}{n} & ... & \frac{1}{n} \\ \frac{1}{n} & -\frac{n-1}{n} & \frac{1}{n} & \frac{1}{n} & ... & \frac{1}{n} \\ \frac{1}{n} & \frac{1}{n} & -\frac{n-1}{n} & \frac{1}{n} & ... & \frac{1}{n} \\ ... & ... & ... & ... & ... & ... \\ \frac{1}{n} & \frac{1}{n} & \frac{1}{n} & \frac{1}{n} & ... & -\frac{n-1}{n} \end{bmatrix}, \end{aligned}$$

$$
\begin{aligned}
M(\boldsymbol{\nabla}_{\boldsymbol{s}}^{\dagger}) &= \sqrt{\frac{(n-1)^2}{n^2} + \frac{1}{n^2} \times (n-1)} \\
&= \sqrt{\frac{n^2 - 2n + 1 + n - 1}{n^2}} \\
&= \sqrt{\frac{n-1}{n}}.
\end{aligned}
$$

When $n \to \infty$, $M(\boldsymbol{\nabla}_{\boldsymbol{s}}^{\dagger}) \to 1$. □

Lemma 2.6.3. *For a Chain Sparsifier with n nodes and $n-1$ edges, $M = \sqrt{\frac{n}{4}}$ when n is even and $M = \frac{1}{2}\sqrt{n - \frac{1}{n}}$ when n is odd.*

Proof. Without loss of generality, we can assume Chain Sparsifier starts at node 1 and $e_1 = \{1, 2\}$, $e_2 = \{2, 3\}$,...,$e_{n-1} = \{n-1, n\}$.

By definition, $\boldsymbol{\nabla}_{\boldsymbol{s}} \in R^{(n-1)\times n}$ and

$$
\boldsymbol{\nabla}_{\boldsymbol{s}} = \begin{bmatrix}
1 & -1 & 0 & 0 & ... & 0 \\
0 & 1 & -1 & 0 & ... & 0 \\
0 & 0 & 1 & -1 & ... & 0 \\
... & ... & ... & ... & ... & ... \\
0 & 0 & 0 & 0 & ... & -1
\end{bmatrix},
$$

$$
\boldsymbol{\nabla}_{\boldsymbol{s}}\boldsymbol{\nabla}_{\boldsymbol{s}}^{\boldsymbol{T}} = \begin{bmatrix}
2 & -1 & 0 & 0 & ... & 0 \\
-1 & 2 & -1 & 0 & ... & 0 \\
0 & -1 & 2 & -1 & ... & 0 \\
... & ... & ... & ... & ... & ... \\
0 & 0 & 0 & 0 & ... & -1
\end{bmatrix},
$$

$$(\boldsymbol{\nabla_s}\boldsymbol{\nabla_s^T})^{-1} = \begin{bmatrix} \frac{n-1}{n} & \frac{n-2}{n} & \frac{n-3}{n} & \frac{n-4}{n} & \dots & \frac{1}{n} \\ \frac{n-2}{n} & \frac{2(n-2)}{n} & \frac{2(n-3)}{n} & \frac{2(n-4)}{n} & \dots & \frac{2}{n} \\ \frac{n-3}{n} & \frac{2(n-3)}{n} & \frac{3(n-3)}{n} & \frac{3(n-4)}{n} & \dots & \frac{3}{n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{1}{n} & \frac{2}{n} & \frac{3}{n} & \frac{4}{n} & \dots & \frac{n-1}{n} \end{bmatrix},$$

Therefore,

$$\begin{aligned} \boldsymbol{\nabla_s^\dagger} &= \boldsymbol{\nabla_s^T}(\boldsymbol{\nabla_s}\boldsymbol{\nabla_s^T})^{-1} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 & ... & 1 \\ -1 & 0 & 0 & 0 & ... & 0 \\ 0 & -1 & 0 & 0 & ... & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & ... & -1 \end{bmatrix} \begin{bmatrix} \frac{n-1}{n} & \frac{n-2}{n} & \frac{n-3}{n} & \frac{n-4}{n} & \dots & \frac{1}{n} \\ \frac{n-2}{n} & \frac{2(n-2)}{n} & \frac{2(n-3)}{n} & \frac{2(n-4)}{n} & \dots & \frac{2}{n} \\ \frac{n-3}{n} & \frac{2(n-3)}{n} & \frac{3(n-3)}{n} & \frac{3(n-4)}{n} & \dots & \frac{3}{n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{1}{n} & \frac{2}{n} & \frac{3}{n} & \frac{4}{n} & \dots & \frac{n-1}{n} \end{bmatrix} \\ &= \begin{bmatrix} \frac{n-1}{n} & \frac{n-2}{n} & \frac{n-3}{n} & \frac{n-4}{n} & \dots & \frac{1}{n} \\ -\frac{1}{n} & \frac{n-2}{n} & \frac{n-3}{n} & \frac{n-4}{n} & \dots & \frac{1}{n} \\ -\frac{1}{n} & -\frac{2}{n} & \frac{n-3}{n} & \frac{n-4}{n} & \dots & \frac{1}{n} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ -\frac{1}{n} & -\frac{2}{n} & -\frac{3}{n} & -\frac{4}{n} & \dots & -\frac{n-1}{n} \end{bmatrix}. \end{aligned}$$

When n is even

$$M(\boldsymbol{\nabla}_{\boldsymbol{s}}^{\dagger}) = \sqrt{\frac{1}{4} \times n} = \sqrt{\frac{n}{4}}.$$

When n is odd

$$\begin{aligned} M(\boldsymbol{\nabla}_{\boldsymbol{s}}^{\dagger}) &= \sqrt{(\frac{(n+1)/2}{n})^2 \times (n-1)/2 + (\frac{(n-1)/2}{n})^2 \times (n+1)/2} \\ &= \sqrt{\frac{(n+1)^2/4 \times (n-1)/2}{n^2} + \frac{(n-1)^2/4 \times (n+1)/2}{n^2}} \\ &= \frac{1}{2}\sqrt{n - \frac{1}{n}}. \end{aligned}$$

□

For MST sparsifiers, $\boldsymbol{\nabla}_{\boldsymbol{s}} \in \mathcal{R}^{(n-1)\times n}$ and the proof is the same as the one for Chain Sparsifier. Also, from observations, $M_{Uniform} < M_{KN_w} < M_{KN}$.

2.7 Application on Email Data

For the real data application, the network was generated using email data from a large European research institution. We have anonymous information about all incoming and outgoing email between members of the research institution. There is an edge (u, v) in the network if person u sent person v at least one email. The emails only represent communication between institution members (the core), and the dataset does not contain incoming messages from or outgoing messages to the rest of the world.

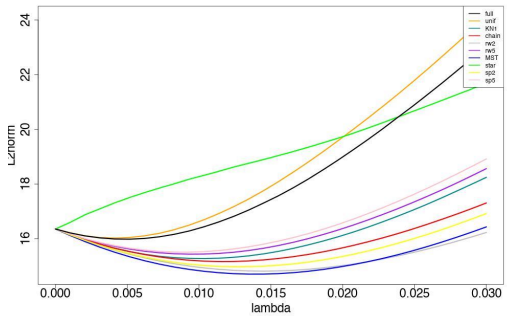
The network contains 1005 nodes, 25000 edges, with 42 clusters embedded into

Methods	Number of Edges					
	1k		2k		3k	
	Time (Seconds)					
	Spar.	Comp.	Spar.	Comp.	Spar.	Comp.
Full	0	44.88	0	44.88	0	44.88
MST	0.397	4.512	0.789	5.542	1.244	7.241
Chain	0.002	4.235	0.026	4.811	0.003	5.502
RW2	0.321	4.089	0.620	5.171	0.979	6.640
RW5	0.316	5.105	0.644	6.706	0.973	7.823
Star	0.001	4.303	0.001	5.232	0.001	5.556
KN1	0.274	4.307	0.275	5.532	0.283	6.019
Unif	0.001	4.417	0.001	5.425	0.001	6.141
Span2	0.356	4.230	1.000	5.379	1.086	7.403
Span5	0.320	5.344	0.621	7.155	0.934	8.378

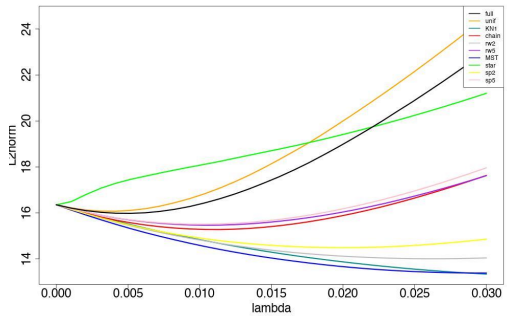
Table 2.3: Sparsification time(s) and computational time(s) for different edges.

it. We removed the duplicated edges and modified the self-loops. The original graph is directed, but we changed it to an undirected one. We define the observed value as $y_i = \beta_i + \epsilon_i$, where $\beta_i \in [1, 42]$ and $\epsilon_i \sim N(0, 0.5)$.

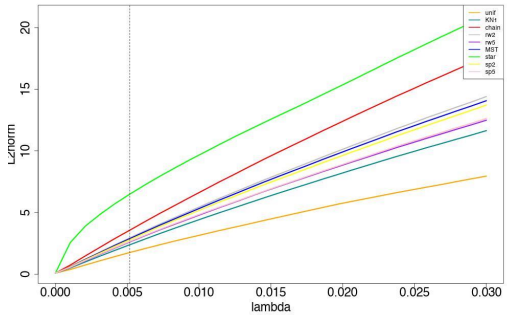
From Figure 2.23 and Figure 2.24, we show the estimation results from unweighted and weighted sparsifiers, as well as the sparsity. We can see that because of the noise, estimation from many sparsifiers are actually better than estimation from the full graph. This shows that the original graph is noisy and sparsifiers can eliminate most of the noise from the graph. Also, weighted sparsifiers perform better than unweighted ones. MST and KN have the best performances, followed by Random Walk and Uniform. The sparsities of MST and KN are also better than the rest of the sparsifiers. From Table 2.3, we can see that computational time has been significantly reduced. For sparsifiers with 1000 edges, computational time is reduced by nearly 90 percent.



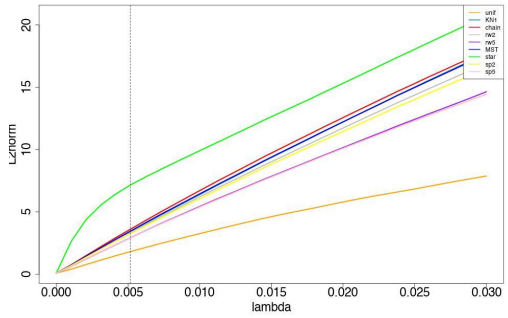
(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (Unweighted).



(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \boldsymbol{\beta}_{true}||_2$ vs. λ (Weighted).

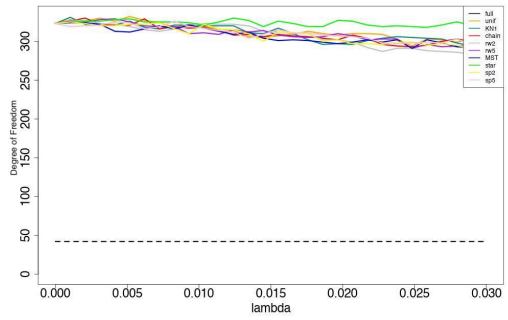


(a) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (Unweighted).

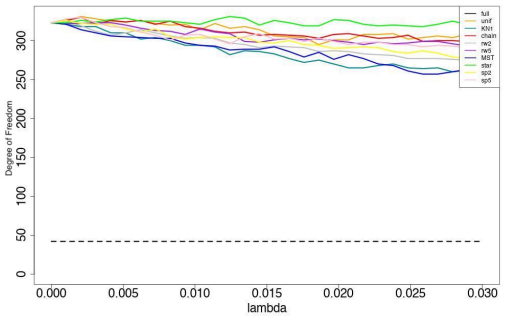


(b) $||\hat{\boldsymbol{\beta}}_{\tilde{\boldsymbol{G}}} - \hat{\boldsymbol{\beta}}_{\boldsymbol{G}}||_2$ vs. λ (Weighted).

Figure 2.23: RSS of weighted and unweighted sparsifiers for email dataset.



(a) Sparsity for unweighted sparsifiers



(b) Sparsity for weighted sparsifiers

Figure 2.24: Sparsity for weighted and unweighted sparsifiers.

2.8 Discussion

Solving a regression problem on a network structure is a popular problem. When there is a grouping structure within the network, one way to solve it is to add

a penalty term on the deviation of parameter values, making it a Fused LASSO problem. The biggest challenge when solving this problem is the computational burden. When the graph is dense, computational time grows quadratically with the number of nodes. We propose a method of finding a sparsified graph based on the original one, and solve the regression problem on the sparsified graph instead. We show that under certain graph structures, working with a sparsified graph is just as effective as that of the original graph. In fact, there are cases where a sparsified graph outperforms the original ones. We explore different factors that affect the performances, including types of graphs, sparsifiers, number of edges, weights, and etc. We also show that a sparsified graph works better than Community Detection algorithms, especially when there is large noise and more than 10 clusters in the graph.

Chapter 3

Decentralized Federated Learning and Knowledge Transfer

3.1 Introduction

Federated Learning is a novel approach that has gained popularity in recent years. The concept of Federated Learning was first introduced in 2016. In this paper, McMahan et al. (2017) introduced the learning of deep neural networks from decentralized data, which significantly improves communication efficiency. The main idea of Federated Learning is to allow models to train across different sites or locations without the need to share raw data with each other. This method focuses on keeping data localized, protecting user's privacy, and reducing the need for excessive data transmission to a centralized server. In Federated Learning, every site performs computation on its own data and shares the model or model parameters afterwards. Some examples include mobile phone data or hospital data, where each site generates a local model, with model updates being sent to a central server. The central server aggregates information from each site into a global model and

then sends model updates back to each individual site. In this way, data privacy is preserved, and network load is significantly reduced. Federated Learning has applications in many different areas. Hard et al. (2018) utilize Federated Learning to make mobile keyboard predictions. The goal is to check whether people click on the recommended links. Wang et al. (2022) introduce a mobility digital twin (MDT) framework, which is a data-driven framework for mobility services based on AI algorithms. This framework aims to integrate humans, vehicles, and traffic signals together. In healthcare, Kaissis et al. (2020) leverage AI algorithms to share medical imaging data while maintaining accuracy. In agriculture, Durrant et al. (2022) develop a cross-silo Machine Learning algorithm that helps with data sharing across the supply chain. In particular, the goal is to optimize soybean yield prediction through Federated Learning by training models without sharing raw data. Additionally, Saputra et al. (2019) introduce a federated energy demand learning approach to predict energy demand for electric vehicle (EV) networks.

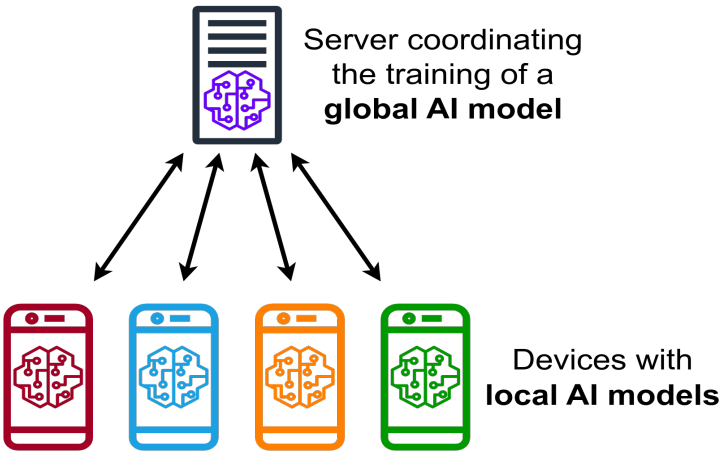


Figure 3.1: Federated learning example.

The biggest advantage of Federated Learning, as mentioned above, is its privacy preservation. Since data remains on the original device, privacy risks are significantly

lower. At the same time, since only model parameters or model updates are being transmitted, communication overhead is also reduced.

Centralized Federated Learning is a hot topic in Machine Learning literature. Over the past few years, many Centralized Federated Learning frameworks have been proposed and proven effective in many different fields. Li et al. (2020) propose a framework where the central server learns to detect and remove malicious models and lead to targeted defense. This framework has proven effective in classification and sentiment analysis tasks. Corinzia and Buhmann (2019) propose a VIRTUAL algorithm that treats the network of the server and the clients as a star-shaped Bayesian network, where the central server is the center of the star. Learning is performed on the network using approximated variational inference, which has been shown to be effective on real-world federated datasets. Le et al. (2021) present a Federated Continuous Learning scheme based on Broad Learning (FCL-BL) with a central server to support efficient and accurate Federated Continuous Learning (FCL), which effectively solves the problem of one-time learning without consideration for continuous learning. Chen et al. (2019) present an Asynchronous Online Federated Learning (ASO-Fed) framework, which updates the central model in an asynchronous manner to deal with varying computational costs at heterogeneous edge devices and devices that lag behind. Nilsson et al. (2018) compare centralized analysis with three Federated Learning algorithms on IID MNIST and non-IID MNIST data. The result shows that they are equivalent on IID data, but the centralized approach outperforms FedAvg with non-IID data.

However, some potential problems still exist. First of all, managing communication between devices and the central server can be costly. Although Centralized Federated Learning has significantly reduced communication costs compared to traditional methods by transferring only model parameters, when there are a large

number of clients in parallel, communication resources can still be problematic. Numerous methods have been proposed to cope with this problem. For instance, we can either learn from a restricted space from a smaller number of variables or compress the model so that model sizes become smaller when transmitting (Konečný et al., 2016). Sattler et al. (2019) extend the existing compression methods by proposing a Sparse Ternary Compression (STC) framework, which broadens the utility in Federated Learning settings.

Computational and storage resources are another problem. In current Federated Learning scenarios, the number of clients can be as large as one billion. The central server needs to store and aggregate models from all clients, which can easily exceed the capacity of it. One potential solution is to use lightweight models instead. This can be done by either filtering features in some of the model layers (Sandler et al., 2018) or scaling all dimensions of the models using effective compound coefficients (Tan and Le, 2019).

In addition, fairness and trust are big issues in Centralized Federated Learning. Questions have arisen on topics such as the performance and validity of the global model, as well as the security and privacy of the central server. Many recent research efforts focus on different questions. Bagdasaryan et al. (2020) show that anyone can backdoor a Federated Learning algorithm and cause the global model to reach an unrealistically high accuracy. Tolpegin et al. (2020) study the problem of data poisoning attacks by malicious participants, among others.

Lastly, dealing with non-IID data is another issue. In Federated Learning, training a global model utilizing all clients' data becomes significantly harder when clients' data is non-IID, which is caused by four different reasons. The first one is covariate shift, which means the marginal distribution of X for each client is different, even when the conditional distribution of y given X is the same. The

second is prior probability shift, which means the marginal distribution of y is different, even when the conditional distribution of X given y is the same. The third is concept drift, which means for each client, the conditional distribution of y given X is different. The last one is unbalancedness, which means different clients hold different numbers of data. To solve these problems, a number of algorithms have been proposed. K.Tam et al. (2023) propose FedMix to deal with clients with mislabeled data; Wicaksana et al. (2022) introduces FedNCI for clients with different noise distribution; Li et al. (2022) propose ARFL for clients with data corruption of non-IID data; Xia et al. (2021) propose Auto-FedAvg for non-IID distribution of data, among others.

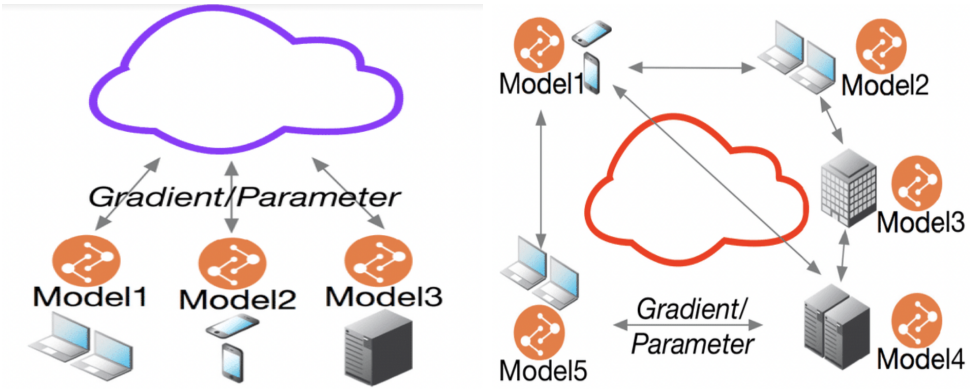


Figure 3.2: Centralized vs. Decentralized Federated Learning.

Decentralized Federated Learning is another structure in which clients or sites communicate with each other without the need for a central server. This concept was initially proposed in 2019 by Lalitha et al. (2018). In this paper, the authors introduce the concept of Decentralized Federated Learning and its potential applications in neural network algorithms. The biggest advantage of Decentralized Federated Learning is that it removes the central server, which can result in huge savings in communication costs and resources.

Current research on Decentralized Federated Learning has different focuses. In-

formation exchange is one of the most important ones. Jiang and Hu (2020) introduce the adaptive partial gradient aggregation method. This method employs a partial gradient exchange mechanism coupled with an adaptive model updating method, optimizing the convergence rate and significantly reducing training time without compromising prediction accuracy. Li and Chen (2021) propose a decentralized zeroth-order method based on biased stochastic zeroth-order updates, which notably reduces the number of communication rounds needed for convergence. Ye et al. (2021) propose the Soft-DSGD method, which updates model parameters with partially received messages and optimizes mixing weights according to the link reliability matrix, achieving convergence rates comparable to those with perfect communications. Wu et al. (2022) present a novel Gradient Descent algorithm focusing on the impact of the learning rate and network structure on Decentralized Federated Learning.

Communication protocol is another hot topic. Many studies focus on how peers communicate with each other. One of the most common strategies is one-to-one communication. The Gossip protocol is the most well-known algorithm that falls into this category (Boyd et al., 2006). Essentially, at each iteration, each client randomly finds another client to connect with and shares their model weights with each other to update their own models. At the same time, there is a broadcast protocol, which lets the clients communicate in a one-to-many manner. This method was initially proposed by Nedic (2010), and the asynchronous broadcast-based algorithm is applied to networks subject to random link failures. Hybrid protocols are developed by combining the two above. This means the communication can be one-to-one or one-to-many, depending on the structure of the network. This idea was proposed by Aysal et al. (2009) as a peer-to-neighbor approach, where the client first communicates with its multiple neighbors before engaging in one-to-one

communication.

Besides, how to aggregate model updates is a controversial topic. The traditional paradigm is to let each client receive models from other clients and aggregate them into their own local model by taking weighted averages. On the other hand, the continual paradigm proposes that each client receives one model from a peer for every iteration. Each client trains that model on their local data, and passes the model to the next client. In this way, learning is highly accurate and personalized. As examples, Assran et al. (2019) and Shi et al. (2021) adopt the aggregate paradigm. Assran et al. (2019) compares the hybrid and broadcast protocols and Shi et al. (2021) shows the convergence of hybrid protocols in a network. Sheller et al. (2019) adopts the continual paradigm and shows how to output a model under decentralized approaches.

Lastly, network topology is commonly seen in many Decentralized Federated Learning papers. In essence, network topology defines how knowledge is disseminated among clients. Nedić et al. (2018) give a comprehensive review of different network topologies, including line, ring, mesh, star, tree, and etc. Line and ring imply that knowledge is transferred one by one in a certain order. Mesh means the network is fully connected and anyone is allowed to communicate with anyone. Star means there is a client that acts as the "central server" and knowledge is always transferred from the center to other clients, or from other clients to the center. For instance, Pappas et al. (2021) propose an interplanetary file system (IPFS) framework based on a star topology. Chen et al. (2022) adopt the mesh topology, proposing a decentralized wireless Federated Learning algorithm called DWFL that significantly reduces communication rounds.

In this section, we aim to discuss the following two problems: how do different information exchange methods affect prediction accuracy? How do different network

topologies affect training efficiency?

3.2 Notations

In the context of Decentralized Federated Learning, N represents the total number of sites engaged in the learning process, each indexed by j, where j ranges from 1 to N. At each site j, $\boldsymbol{y}_j$ denotes the vector of target variables, and $\boldsymbol{X}_j$ signifies the matrix of feature variables, with each row corresponding to an observation and each column to a feature. The vector $\boldsymbol{\beta}_j$ refers to the model parameters that are specific to site j. The weight α_j is assigned to the objective function of site j, reflecting the relative size or importance of its dataset in the aggregated learning process. The function $f(\boldsymbol{y}_j, \boldsymbol{X}_j, \boldsymbol{\beta}_j)$ represents the local objective function at site j, which measures the loss or error of the model on the local dataset, contributing to the formulation of the overall learning objective across the decentralized network.

The global objective function is defined as a weighted sum of the local objective functions, aimed at minimizing the overall loss across all sites.

$$\min_{\boldsymbol{\beta_1}, \boldsymbol{\beta_2}, \ldots, \boldsymbol{\beta_N}} \left(\sum_{j=1}^{N} \alpha_j f(\boldsymbol{y_j}, \boldsymbol{X_j}, \boldsymbol{\beta_j}) \right). \tag{3.1}$$

The goal of the Decentralized Federated Learning process is to determine the optimal set of model parameters for each site. This involves each site computing updates to its local model parameters $\boldsymbol{\beta_j}$ based on its data $(\boldsymbol{y_j}, \boldsymbol{X_j})$, followed by a collaborative mechanism among parameters of neighboring sites. The process may involve iterative communication and updates until convergence is achieved.

3.3 Decentralized Federated Learning

3.3.1 Local Estimation

Local estimation methods are popular because they are simple in concept and implementation and can be set up quickly, making them a good starting point for many problems. Specifically, for a specific node j:

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{j}} = \min_{\boldsymbol{\beta}_j \in \mathbf{R}^d} f(\boldsymbol{y}_j, \boldsymbol{X}_j, \boldsymbol{\beta}_j). \tag{3.2}$$

From its name, local estimation focuses on analyzing data from their own individual parts of a network or system. One big advantage is that each estimation is independent of the rest. Therefore, if something goes wrong with the data in one part, it doesn't affect the rest. Also, this method is very intuitive and easy to implement.

However, a significant downside is that local estimations do not take into account how different parts of the system or network might be connected. This means they might miss out on important information from other parts if data is heterogeneous or if the distribution varies among parts. Additionally, if the individual data size is small, local estimation can easily overfit or fit a model that is not representative of the entire population.

In summary, although local estimation is simple and easy to set up, it is often not the best method to use because of its inability to capture the full picture. When connections between networks are important, local estimations cannot leverage information from neighbors, leading to weak model predictions.

3.3.2 Data Aggregation

In the approach known as data aggregation, when looking at a specific part of a system or network, we assume that it has the same parameters as its connected parts or the rest of the system. For example, if we're trying to solve for certain parameter values for one part of the network, we assume these parameter values are the same for its connected neighbors. Because of this assumption, to find the best values for these parameters, we combine the data from all connected parts and look for the ones that fit the best.

This implies that for a specific node j, we assume $\boldsymbol{\beta_j} = \boldsymbol{\beta_i}$, $\forall (i,j) \in \boldsymbol{E}$. Given $(\boldsymbol{y_i}, \boldsymbol{X_i})$ for all i:

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{j}} = \min_{\boldsymbol{\beta}_j \in \mathbf{R}^d} \sum_{(i,j) \in \boldsymbol{E}} f(\boldsymbol{y_i}, \boldsymbol{X_i}, \boldsymbol{\beta}_j). \tag{3.3}$$

There are a lot of benefits to this method. Firstly, by pooling data together from different parts, we significantly increase our data size. Therefore, the results will be more robust and representative of the entire population. At the same time, this method simplifies the problem by assuming that every site shares the same set of parameters. With this assumption, the model complexity is very low, making it very unlikely that the model will overfit.

However, the assumption of similarity is also the biggest drawback. We assume that the data from all connected parts of the network are similar enough to be treated as coming from the same source. This is often not true for many Federated Learning problems, where each site has its own data source and they do not necessarily share the same parameters. At the same time, data privacy is crucial for many problems. The goal is to keep data local and not share it across the network. Data aggregation goes against this principle by combining data from different parts.

3.3.3 Sharing Gradient

In a Decentralized Federated Learning setting, when multiple sites within a network share information with each other, one feasible method is to utilize gradients from its own data as well as those from the neighbors. Specifically, for a given node at iteration t, the parameter update is determined by the weighted sum of gradients from the data of connected nodes.

For a specific node j, at the t-th iteration,

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{j}}^{(\boldsymbol{t})} = \hat{\boldsymbol{\beta}}_{\boldsymbol{j}}^{(\boldsymbol{t-1})} + \lambda \left(\frac{\partial}{\partial \boldsymbol{\beta}_j} f(\boldsymbol{y}_j, \boldsymbol{X}_j, \hat{\boldsymbol{\beta}}_{\boldsymbol{j}}^{(\boldsymbol{t-1})}) + \sum_{(i,j)\in \boldsymbol{E}} \alpha_i \frac{\partial}{\partial \boldsymbol{\beta}_i} f(\boldsymbol{y}_i, \boldsymbol{X}_i, \hat{\boldsymbol{\beta}}_{\boldsymbol{i}}^{(\boldsymbol{t-1})}) \right). \quad (3.4)$$

One of the main advantages of this method is its preservation of privacy. This method does not involve sharing raw data or individual models across different sites. Since gradients convey derivative information rather than the direct data, gradients from other sites can contribute to the learning objective without disclosing sensitive or privacy information. At the same time, since gradients point in the directions of improvement, a combined gradient represents a unified direction of improvement that is applicable across diverse datasets.

Nevertheless, this method holds the same challenges as sharing models. Determining the optimal weights for gradients from different nodes can be complicated. Without much prior knowledge about each site, it can be hard to attain optimal convergence speed and objective values.

For example, assume there are two sites only. Pseudo-code is shown in Algorithm 6.

Algorithm 6 Sharing Gradient

procedure SHAREGRADIENTS($\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, T, λ_1, λ_2)
  Input: Initial parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, total iterations T, learning rates λ_1, λ_2
  Output: Updated parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
  for $t = 1$ to T **do**
    if $t \mod 2 \neq 0$ **then**
      Compute gradient at $\boldsymbol{\beta}_1^{(t-1)}$: $\nabla f_1 = \frac{\partial}{\partial \boldsymbol{\beta}_1} f(\boldsymbol{\beta}_1^{(t-1)})$
      Compute gradient at $\boldsymbol{\beta}_2^{(t-1)}$: $\nabla f_2 = \frac{\partial}{\partial \boldsymbol{\beta}_2} f(\boldsymbol{\beta}_2^{(t-1)})$
      Update $\boldsymbol{\beta}_1^{(t)} \leftarrow \boldsymbol{\beta}_1^{(t-1)} - \lambda_1(\nabla f_1 + \lambda_2 \nabla f_2)$
    else
      Compute gradient at $\boldsymbol{\beta}_2^{(t-1)}$: $\nabla f_2 = \frac{\partial}{\partial \boldsymbol{\beta}_2} f(\boldsymbol{\beta}_2^{(t-1)})$
      Compute gradient at $\boldsymbol{\beta}_1^{(t-1)}$: $\nabla f_1 = \frac{\partial}{\partial \boldsymbol{\beta}_1} f(\boldsymbol{\beta}_1^{(t-1)})$
      Update $\boldsymbol{\beta}_2^{(t)} \leftarrow \boldsymbol{\beta}_2^{(t-1)} - \lambda_1(\nabla f_2 + \lambda_2 \nabla f_1)$
    end if
  end for
  return $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
end procedure

3.3.4 Model Fusion

For this particular method, we minimize the function that evaluates the model's fit to its own data ($\boldsymbol{y_j}$, $\boldsymbol{X_j}$) while also integrating a regularization term. This term imposes a penalty on the absolute value of the difference between parameter estimates of node j and those of its neighboring nodes, and sums them together. The effect of the penalty term is controlled by a coefficient λ, which determines how significantly the deviations in parameter values affect the overall objective function.

Specifically, for a specific node j,

$$\hat{\boldsymbol{\beta}}_{\boldsymbol{j}} = \min_{\boldsymbol{\beta}_j \in \mathbf{R}^d} f(\boldsymbol{y}_j, \boldsymbol{X}_j, \boldsymbol{\beta}_j) + \lambda \sum_{(i,j) \in E} |\boldsymbol{\beta}_j - \hat{\boldsymbol{\beta}}_{\boldsymbol{i}}|. \tag{3.5}$$

This method borrows ideas from Fused LASSO and Gossip averaging. Boyd et al. (2005) introduces the concept of Gossip averaging, which involves how peers

in a network can share information with each other without the need for a central server. In this case, individual model parameters are shared with each other. With information from other sites being aggregated, each site forms a fused LASSO problem.

This method assumes similarity among peers that are connected to each other, and works well when the network structure has strong relevance to the system being analyzed, including social, sensor, or biological networks. However, the homogeneity assumption can be potentially problematic as the assumption of similarity between connected nodes may not always hold. When a network is dense or dynamically changing, each node can have connections that are not necessarily informative.

For two sites, below is the pseudo code in Algorithm 7. Note that

$$\nabla|\boldsymbol{\beta_1} - \boldsymbol{\beta_2}| = \left[\frac{\partial|\boldsymbol{\beta_{1,i}} - \boldsymbol{\beta_{2,i}}|}{\partial\boldsymbol{\beta_{1,i}}}\right]_{i=1}^{n},$$

where

$$\frac{\partial|\boldsymbol{\beta_{1,i}} - \boldsymbol{\beta_{2,i}}|}{\partial\boldsymbol{\beta_{1,i}}} = \begin{cases} -1 & \text{if } \boldsymbol{\beta_{2,i}} > \boldsymbol{\beta_{1,i}}, \\ 1 & \text{if } \boldsymbol{\beta_{2,i}} < \boldsymbol{\beta_{1,i}}, \\ [\text{-}1,1] & \text{if } \boldsymbol{\beta_{2,i}} = \boldsymbol{\beta_{1,i}}. \end{cases}$$

3.3.5 Mutual Knowledge Transfer

This method is adapted from the paper "Decentralized Federated Learning via Mutual Knowledge Transfer". Li et al. (2021) introduce a method to enhance Decentralized Federated Learning in IoT systems. The idea is to share model parameters between sites. At each iteration, we select random pairs of sites for communication. Within each pair, one site is the sender and the other is the receiver. The sender

Algorithm 7 Fusion in Federated Learning

procedure FUSION($\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, T, λ_1, λ_2)
  Input: Initial parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, total iterations T, learning rates λ_1, λ_2
  Output: Updated parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
  for $t = 1$ to T **do**
    if $t \mod 2 \neq 0$ **then**
      Compute the gradient for $\boldsymbol{\beta}_1^{(t-1)}$: $\nabla f_1 = \frac{\partial}{\partial \boldsymbol{\beta}_1} f(\boldsymbol{\beta}_1^{(t-1)})$
      Compute the gradient of the penalty term: $\nabla P_{12} = \nabla |\boldsymbol{\beta}_1^{(t-1)} - \boldsymbol{\beta}_2^{(t-1)}|$
      Update $\boldsymbol{\beta}_1^{(t)} \leftarrow \boldsymbol{\beta}_1^{(t-1)} - \lambda_1(\nabla f_1 + \lambda_2 \nabla P_{12})$
    else
      Compute the gradient for $\boldsymbol{\beta}_2^{(t-1)}$: $\nabla f_2 = \frac{\partial}{\partial \boldsymbol{\beta}_2} f(\boldsymbol{\beta}_2^{(t-1)})$
      Compute the gradient of the penalty term: $\nabla P_{21} = \nabla |\boldsymbol{\beta}_2^{(t-1)} - \boldsymbol{\beta}_1^{(t-1)}|$
      Update $\boldsymbol{\beta}_2^{(t)} \leftarrow \boldsymbol{\beta}_2^{(t-1)} - \lambda_1(\nabla f_2 + \lambda_2 \nabla P_{21})$
    end if
  end for
  return $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
end procedure

site takes its model to the receiver site, and gets prediction results using the receiver site's data. Both sites get their parameters updated through cross entropy loss and the Kullback-Leibler (KL) divergence between the two predictions.

The novelty of this method comes from the incorporation of a KL divergence term into the objective function. This term penalizes the discrepancies in predictions between different sites. When combined with cross entropy loss, this algorithm guides models towards better generalization across diverse datasets.

Suppose there are K clients in the network, and each possesses a private dataset $\mathcal{D}_k = \{(x_i^k, y_i^k)\}_{i=1}^{N_k}$ where N_k is the number of data points for client k, $k = 1, \ldots, K$, x_i^k is the i-th data point of the k-th client, and $y_i^k \in \{1, 2, \ldots, C\}$ is the corresponding label. Now, at the beginning of the t-th round, S clients, with indices denoted as $\mathcal{I}_t^A = \{k_t^1, \ldots, k_t^S\}$, are randomly selected to communicate with another S different clients, indices denoted by $\mathcal{I}_t^B = \{k_t^{S+1}, \ldots, k_t^{2S}\}$ in the dataset in a one-to-one manner, with S being less than half the number of clients. Then, each of the clients

in $\mathcal{I}_t^A$ updates its local model $\boldsymbol{\beta}_t^{k_t^j}$ over its private data $\mathcal{D}_{k_t^j}$ through Stochastic Gradient Descent with local minibatch size B_1 and learning rate η_0 by M passes:

$$\tilde{\boldsymbol{\beta}}_t^{k_t^j} = \mathrm{SGD}_{B_1,M}(\boldsymbol{\beta}_t^{k_t^j}, \mathcal{D}_{k_t^j}), \forall j = 1, \ldots, S.$$

Then, each of the clients from $\mathcal{I}_t^A$ transfers their updated model to the corresponding client in $\mathcal{I}_t^B$. Namely, the k_t^j-th client transfers its model to the k_t^{j+S}-th client, and the k_t^{j+S}-th client mixes both models and transfers knowledge to each other. Specifically, the dataset $\mathcal{D}_{k_t^{j+S}}$ is split into L mini-batches of size B_2 denoted by $\mathcal{B}_l$, $l = 1, \ldots, L$. The k_t^{j+S}-th clients make predictions using their models $\tilde{\boldsymbol{\beta}}_t^{k_t^j}$ and $\boldsymbol{\beta}_t^{k_t^{j+S}}$. Because these two models are trained on different data, their prediction results are different too. The loss function is defined in a way that combines the cross entropy loss and KL divergence between the two model predictions. Specifically,

$$\mathrm{Loss}_1(\tilde{\boldsymbol{\beta}}_t^{k_t^j}, \mathcal{B}_l, \mathcal{P}_{2,l}) = L_C(\mathcal{P}_{1,l}, \mathcal{Y}_l) + D_{KL}(\mathcal{P}_{2,l} \parallel \mathcal{P}_{1,l}), \tag{3.6}$$

and

$$\mathrm{Loss}_2(\boldsymbol{\beta}_t^{k_t^{j+S}}, \mathcal{B}_l, \mathcal{P}_{1,l}) = L_C(\mathcal{P}_{2,l}, \mathcal{Y}_l) + D_{KL}(\mathcal{P}_{1,l} \parallel \mathcal{P}_{2,l}), \tag{3.7}$$

where

$$\mathcal{P}_{1,l} = \{\mathbf{p}_{1,l,z}\}_{z=1}^{B_2} = \mathrm{model}(\mathcal{B}_l, \tilde{\boldsymbol{\beta}}_t^{k_t^j}), \forall l,$$
$$\mathcal{P}_{2,l} = \{\mathbf{p}_{2,l,z}\}_{z=1}^{B_2} = \mathrm{model}(\mathcal{B}_l, \boldsymbol{\beta}_t^{k_t^{j+S}}), \forall l,$$

and

$$\mathbf{p}_{1,l,z} = [p_{1,l,z}^1, p_{1,l,z}^2, ..., p_{1,l,z}^C],$$
$$\mathbf{p}_{2,l,z} = [p_{2,l,z}^1, p_{2,l,z}^2, ..., p_{2,l,z}^C].$$

In the loss function,

$$\begin{aligned} L_C(\mathcal{P}_{1,l}) &= -\sum_{z=1}^{B_2} \mathbf{h}^T(y_z^l) log \mathbf{p}_{1,l,z}, \\ L_C(\mathcal{P}_{2,l}) &= -\sum_{z=1}^{B_2} \mathbf{h}^T(y_z^l) log \mathbf{p}_{2,l,z}, \end{aligned} \tag{3.8}$$

where $\mathbf{h}(y)$ is an one-hot vector with all zeros except the y-th elements being one. Also,

$$\begin{aligned} D_{KL}(\mathcal{P}_{1,l}||\mathcal{P}_{2,l}) &= \sum_{z=1}^{B_2}\sum_{c=1}^{C} p_{1,l,z}^c \log \frac{p_{1,l,z}^c}{p_{2,l,z}^c}, \\ D_{KL}(\mathcal{P}_{1,l}||\mathcal{P}_{2,l}) &= \sum_{z=1}^{B_2}\sum_{c=1}^{C} p_{1,l,z}^c \log \frac{p_{1,l,z}^c}{p_{2,l,z}^c}, \end{aligned} \tag{3.9}$$

which represents the KL divergence between the predictions.

One potential problem of this method is the privacy concern. This model is based on the assumption that each site is willing to share its own model with others. At the same time, each site is willing to offer feedback when other sites bring their models in. To alleviate the privacy concerns and add more flexibility to the algorithms, we can control the update directions. This can be either bidirectional, which means both the sender and receiver get the updates, or unidirectional, which means either the sender or the receiver gets the updates.

Below we present the algorithm in simplified version with two sites only. Pseudo code is shown in Algorithm 8.

When dealing with a classification problem with two classes, assuming there are two sites with parameters $\boldsymbol{\beta_1^t}$ and $\boldsymbol{\beta_2^t}$ at the t-th iteration. For a particular sample z in minibatch l, if $y_z = 1$:

$$L_C(\mathcal{P}_{1,l}) = -\log p_{1,l,z}^1 \text{ where } p_{1,l,z}^1 = \frac{1}{1+\exp(-\boldsymbol{\beta_1^{tT}} x)}.$$

Algorithm 8 Mutual Knowledge Transfer

procedure MKT($\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, T, M, N, D_1, D_2, λ_1, λ_2)
  Input: Initial parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$, total iterations T
    Number of SGD passes M, number of minibatches N
    Datasets D_1 and D_2, learning rates λ_1, λ_2
  Output: Updated parameters $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
  for t `in` $1:T$ **do**
    if $t \mod 2 \neq 0$ **then**
      $\boldsymbol{\beta}_1^{(t)} = \mathrm{SGD}_M(\boldsymbol{\beta}_1^{(t-1)}, D_1)$
      for i `in` $1:N$ minibatches **do**
        $\mathcal{P}_{1i} = \{\mathbf{p}_{1,i,z}\}_{z=1}^{B_2} = \mathrm{predict}(D_2^{(i)}, \boldsymbol{\beta}_1^{(t)})$
        $\mathcal{P}_{2i} = \{\mathbf{p}_{2,i,z}\}_{z=1}^{B_2} = \mathrm{predict}(D_2^{(i)}, \boldsymbol{\beta}_2^{(t-1)})$
        Compute $L_1, L_2, D_1^{KL}, D_2^{KL}$
        $L_1 = L_C(\mathcal{P}_{1,i}) = -\sum_{z=1}^{B_2} \mathbf{h}^T(y_z^i) \log \mathbf{p}_{1,i,z}$
        $L_2 = L_C(\mathcal{P}_{2,i}) = -\sum_{z=1}^{B_2} \mathbf{h}^T(y_z^i) \log \mathbf{p}_{2,i,z}$
        $D_1^{KL} = D_{KL}(\mathcal{P}_{2,i} \parallel \mathcal{P}_{1,i}) = \sum_{z=1}^{B_2} \sum_{c=1}^{C} p_{2,i,z}^c \log \frac{p_{2,i,z}^c}{p_{1,i,z}^c}$
        $D_2^{KL} = D_{KL}(\mathcal{P}_{1,i} \parallel \mathcal{P}_{2,i}) = \sum_{z=1}^{B_2} \sum_{c=1}^{C} p_{1,i,z}^c \log \frac{p_{1,i,z}^c}{p_{2,i,z}^c}$
        $Loss_1 = L_1 + D_1^{KL}$
        $Loss_2 = L_2 + D_2^{KL}$
        $\boldsymbol{\beta}_1^{(t)} = \boldsymbol{\beta}_1^{(t)} - \lambda_1 \frac{\partial Loss_1}{\partial \boldsymbol{\beta}_1^{(t)}}$
        $\boldsymbol{\beta}_2^{(t)} = \boldsymbol{\beta}_2^{(t-1)} - \lambda_2 \frac{\partial Loss_2}{\partial \boldsymbol{\beta}_2^{(t-1)}}$
      end for
    else
      $\boldsymbol{\beta}_2^{(t)} = \mathrm{SGD}_M(\boldsymbol{\beta}_2^{(t-1)}, D_2)$
      for i `in` $1:N$ minibatches **do**
        $\mathcal{P}_{1i} = \{\mathbf{p}_{1,i,z}\}_{z=1}^{B_2} = \mathrm{predict}(D_1^{(i)}, \boldsymbol{\beta}_1^{(t-1)})$
        $\mathcal{P}_{2i} = \{\mathbf{p}_{2,i,z}\}_{z=1}^{B_2} = \mathrm{predict}(D_1^{(i)}, \boldsymbol{\beta}_2^{(t)})$
        Compute $L_1, L_2, D_1^{KL}, D_2^{KL}$
        $Loss_1 = L_1 + D_1^{KL}$
        $Loss_2 = L_2 + D_2^{KL}$
        $\boldsymbol{\beta}_1^{(t)} = \boldsymbol{\beta}_1^{(t)} - \lambda_1 \frac{\partial Loss_1}{\partial \boldsymbol{\beta}_1^{(t)}}$
        $\boldsymbol{\beta}_2^{(t)} = \boldsymbol{\beta}_2^{(t-1)} - \lambda_2 \frac{\partial Loss_2}{\partial \boldsymbol{\beta}_2^{(t-1)}}$
      end for
    end if
  end for
  return $\boldsymbol{\beta}_1$, $\boldsymbol{\beta}_2$
end procedure

And if $y_z = 0$:

$$L_C(\mathcal{P}_{1,l}) = -\log p^0_{1,l,z} \text{ where } p^0_{1,l,z} = \frac{\exp(-\boldsymbol{\beta}_{\mathbf{1}}^{tT} x)}{1 + \exp(-\boldsymbol{\beta}_{\mathbf{1}}^{tT} x)}.$$

If $y_z = 1$, and we let $p_1 = p^1_{1,l,z}$ then

$$\frac{\partial L_C}{\partial \boldsymbol{\beta}_{\mathbf{1}}^{\boldsymbol{t}}} = -\frac{1}{p_1} p_1(1-p_1)x = -(1-p_1)x.$$

If $y_z = 0$, then

$$\frac{\partial L_C}{\partial \boldsymbol{\beta}_{\mathbf{1}}^{\boldsymbol{t}}} = \frac{1}{1-p_1} p_1(1-p_1)x = p_1 x.$$

Also,

$$D_{KL}(\mathcal{P}_{2,l} || \mathcal{P}_{1,l}) = p^0_{2,l,z} \log \frac{p^0_{2,l,z}}{p^0_{1,l,z}} + p^1_{2,l,z} \log \frac{p^1_{2,l,z}}{p^1_{1,l,z}},$$

and the derivative is

$$\frac{\partial D_{KL}}{\partial \boldsymbol{\beta}_{\mathbf{1}}^{\boldsymbol{t}}} = (p^1_{1,l,z} - p^1_{2,l,z})x.$$

Now, When there are multiple classes, we have parameters $\boldsymbol{\Theta}_{\mathbf{1}}^{\boldsymbol{t}}$ and $\boldsymbol{\Theta}_{\mathbf{2}}^{\boldsymbol{t}}$. For batch l, the loss function is defined as

$$Loss_1(\boldsymbol{\Theta}_{\mathbf{1}}^{\boldsymbol{t}}, \mathcal{B}_l, \mathcal{P}_{2,l}) = L_C(\mathcal{P}_{1,l}, \mathcal{Y}_l) + D_{KL}(\mathcal{P}_{2,l} || \mathcal{P}_{1,l}),$$

where

$$\mathcal{P}_{1,l} = \{\mathbf{p}_{1,l,z}\}_{z=1}^{B_2} = model(\mathcal{B}_l, \boldsymbol{\Theta}_{\mathbf{1}}^{\boldsymbol{t}}),$$

$$\mathbf{p}_{1,l,z} = [p^1_{1,l,z}, p^2_{1,l,z}, ..., p^C_{1,l,z}],$$

and

$$L_C(\mathcal{P}_{1,l}) = -\sum_{z=1}^{B_2} \mathbf{h}^T(y_z^l) log \mathbf{p}_{1,l,z}.$$

For a particular data point z and minibatch l, suppose the jth element of y_z is 1 and θ_j is the jth row of Θ_1^t, then

$$L_C = -\log p_{1,l,z}^j = -\log \frac{exp(z_j)}{\sum_{k=1}^{C} exp(z_k)} \text{ where } z_j = \theta_j^T x.$$

Therefore,

$$\frac{\partial L_C}{\partial \theta_j} = -(1 - p_{1,l,z}^j)x.$$

$$\frac{\partial L_C}{\partial \theta_k} = p_{1,l,z}^k x.$$

For non-binary classification at a particular z and l,

$$D_{KL}(\mathcal{P}_{2,l}||\mathcal{P}_{1,l}) = \sum_{m=1}^{M} p_{2,l,z}^m \log \frac{p_{2,l,z}^m}{p_{1,l,z}^m},$$

and the derivative is

$$\frac{\partial D_{KL}}{\partial \theta_j} = (p_{1,l,z}^j - p_{2,l,z}^j)x.$$

3.4 Optimization Approaches

3.4.1 Gradient Descent

Gradient Descent is an optimization algorithm used across Machine Learning to minimize the loss function associated with a particular model. The fundamental principle behind Gradient Descent is to iteratively adjust the model's parameters to reduce the value of the loss function. By computing the gradient of the loss function, a step is made in the opposite direction of the gradient. In this way, Gradient Descent can find the set of parameters that minimizes the loss, effectively training the model. Specifically, to minimize a loss function L, the algorithm adjusts

the parameter vector $\boldsymbol{\beta}$ iteratively to find the minimum of L. Starting with an initial guess $\boldsymbol{\beta}_0$, the algorithm updates $\boldsymbol{\beta}$ at each iteration as follows:

$$\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta}_{\text{old}} - \alpha \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}_{\text{old}}), \tag{3.10}$$

where α denotes the learning rate, a hyperparameter that controls the size of the steps taken towards the minimum. This update step is repeated until convergence, which is indicated by very small changes in $\boldsymbol{\beta}$ or $L(\boldsymbol{\beta})$.

However, Gradient Descent relies on the entire dataset for each update, which makes it computationally expensive for large datasets. The algorithm of Gradient Descent is shown in Algorithm 9.

Algorithm 9 Gradient Descent

```
1: procedure GradientDescent(L, α)
2:     Input: Loss function L, learning rate α
3:     Output: Optimized parameters β
4:     Initialize parameters β
5:     while convergence criteria not met do
6:         Compute the gradient of L at β: g ← ∇_β L(β)
7:         Update β: β ← β − α · g
8:     end while
9:     return β
10: end procedure
```

3.4.2 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a variant of the traditional Gradient Descent algorithm designed to address the computational challenges posed by large datasets. Instead of calculating the gradient based on the entire dataset at every step, SGD updates the model's parameters using only a small batch of samples at a time. This

modification is described mathematically by the update rule:

$$\boldsymbol{\beta}_{\text{new}} = \boldsymbol{\beta}_{\text{old}} - \alpha \nabla_{\boldsymbol{\beta}} L_{\text{batch}}(\boldsymbol{\beta}_{\text{old}}), \tag{3.11}$$

where α is the learning rate, and $\nabla_{\boldsymbol{\beta}} L_{\text{batch}}$ represents the gradient of the loss function L computed only over the current batch.

This approach significantly reduces the computational burden per iteration, leading to faster cycles. Although this method introduces more noise into the parameter updates due to the variability of data in each batch, this randomness can be beneficial. It helps the algorithm escape local minima and can lead to faster convergence in practice, particularly in complex and high-dimensional loss landscapes. This makes SGD especially suitable for large-scale and high-dimensional data applications. Algorithm 10 shows the pseudo-code of Stochastic Gradient Descent.

Algorithm 10 Stochastic Gradient Descent

procedure StochasticGradientDescent(L, α, D)
    Input: Loss function L, learning rate α, dataset D
    Output: Optimized parameters $\boldsymbol{\beta}$
    Initialize parameters $\boldsymbol{\beta}$
    while convergence criteria not met **do**
        Shuffle dataset D
        for each batch $(x_{\text{batch}}, y_{\text{batch}})$ in D **do**
            Compute gradient for batch: $g \leftarrow \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta}; x_{\text{batch}}, y_{\text{batch}})$
            Update $\boldsymbol{\beta}$: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \cdot g$
        end for
    end while
    return $\boldsymbol{\beta}$
end procedure

3.4.3 Fixed and Decay Step Size

Fixed step size in the context of optimization refers to using a constant learning rate throughout the entire training process. This approach simplifies the implementation of algorithms like Gradient Descent, as the learning rate parameter remains unchanged across iterations. However, the biggest drawback of fixed step size is its inflexibility. Although it can be very effective for problems where the loss function is relatively straightforward, it often runs into convergence problems when the loss function is more complex in landscapes. It does not adapt to the changing gradients or the topology of the loss function, which can lead to gradient overshoot, sub-optimal convergence speeds, or convergence to sub-optimal points. The challenge with a fixed step size is choosing a value that is neither too large nor too small, which will either overshoot the gradient or lead to excessively slow convergence.

Decay step size, also known as learning rate decay, is an adaptive approach that modifies the learning rate over time during the training of models. This method normally starts with a higher learning rate, which helps to approach the vicinity of the minimum at a faster speed. Later on, the learning rate is adjusted to lower rates gradually, which helps to converge to the minimum more precisely. There are various decay schemes in use, such as time-based decay and exponential decay, each with its own strategy for reducing the learning rate according to a predefined schedule. For time-based decay, the learning rate decreases linearly over time. Mathematically, it can be represented by:

$$\alpha = \frac{\alpha_0}{1 + kt},$$

where α_0 is the initial learning rate, k is a decay rate, and t is the time. Exponential

decay reduces the learning rate exponentially:

$$\alpha = \alpha_0 \cdot e^{-kt},$$

where α_0 is the initial learning rate, k is a decay constant, and t denotes the epoch number. Among all the methods, some of them decay a lot faster than others, and the need for decaying speed is case by case depending on the optimization problem. This flexibility helps in solving the issues associated with a fixed step size, providing a more dynamic way to adapt to the loss function's landscape.

3.4.4 Adaptive Moment Estimation (ADAM) Optimization

Adaptive Moment Estimation (ADAM), first proposed by Kingma and Ba (2014), is an advanced optimization algorithm that combines the concepts of momentum and learning rate decay. This method offers an adaptive learning rate for each parameter. It calculates a moving average of the gradients and the squared gradients, and then uses these moments to adjust the learning rates individually for each iteration. Mathematically, the first step is to compute the gradient g_t of the loss function with respect to the parameters at time step t. Then we proceed with the following steps: Update the biased first moment estimate:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t.$$

Update the biased second moment estimate:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2.$$

Correct bias in the first moment:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

Correct bias in the second moment:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

Update the parameters:

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}.$$

The parameters β_1 and β_2 control the decay rates of these moving averages. They are typically set close to 1, such as 0.99 or 0.999, α is the initial learning rate, and ϵ is a small constant added to improve numerical stability. Essentially, ADAM will choose the most suitable step size for each iteration, which allows for larger updates for infrequent parameters and smaller updates for frequent ones. Because of this nature, convergence is stabilized under ADAM, making it robust to the choice of initial step size and reducing the need to search for optimal parameters. As a result, ADAM has gained popularity among Machine Learning problems with large datasets and high-dimensional spaces. Its efficiency and effectiveness demonstrate strong performance in both convergence speed and achieving lower loss values. Algorithm 11 shows the pseudo-code of ADAM.

Algorithm 11 ADAM Optimization

procedure ADAM($L, \alpha, \beta_1, \beta_2, \epsilon$)
  Input: Loss function L, initial learning rate α
    Decay rates β_1 and β_2, stabilizing term ϵ
  Output: Optimized parameters $\boldsymbol{\beta}$
  Initialize parameters $\boldsymbol{\beta}$
  Initialize first moment $m_0 \leftarrow 0$, second moment $v_0 \leftarrow 0$
  Initialize timestep $t \leftarrow 0$
  while convergence criteria not met **do**
    $t \leftarrow t + 1$
    Compute gradient: $g_t \leftarrow \nabla_{\boldsymbol{\beta}} L(\boldsymbol{\beta})$
    Update first moment: $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
    Update second moment: $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
    Correct bias in first moment: $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
    Correct bias in second moment: $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
    Update parameters: $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
  end while
  return $\boldsymbol{\beta}$
end procedure

3.5 Transmission Topology

We study transmission topology to answer the question: How can knowledge dissemination be made more efficient? There are different ways of constructing the network, and sites will communicate based on the network structure.

3.5.1 Fully Connected Network

For a fully connected network, we assume every node is connected to every other node in the network. At every iteration, any pair of nodes can transfer information to each other. An example of a fully connected network is shown in Figure 3.3. Regarding the communication algorithm, for each epoch, we randomly select n pairs of sites to update, with one of them being the sender and the other being the receiver. Knowledge gets transferred from sender to receiver. The detailed algorithm is shown

in Algorithm 12.

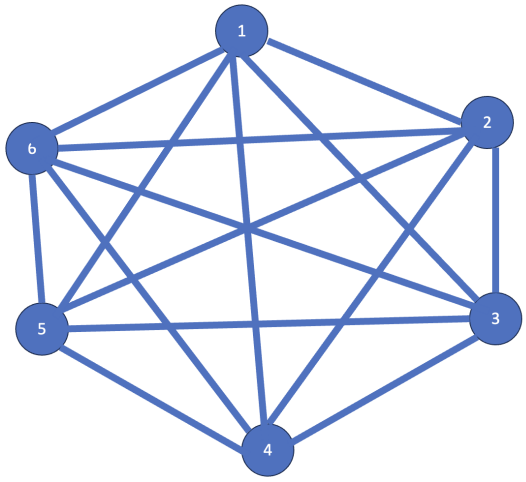


Figure 3.3: Fully connected network illustration.

Algorithm 12 Random Update

```
procedure RandomUpdate(T, n_sites, n, β)
    Input: Total epochs T, number of sites n_sites, number of pairs n
           Parameter matrix β with vectors for each site
    Output: Updated parameter matrix β
    Initialize index list: ind ← list(range(n_sites))
    for t = 1 to T do
        for i = 1 to n do
            Shuffle index list: random.shuffle(ind)
            Select the first two unique indices for model transfer
            Call MKT for model transfer: MKT(β[ind[0]], β[ind[1]])
        end for
    end for
    return β
end procedure
```

3.5.2 Chain Network

Chain means that we randomly select n sites and connect them into a chain. The updating order is determined by the chain that is randomly generated in each iter-

ation. For each epoch, information sharing always starts from the head of the chain and gradually propagates to the tail of it. An illustration of this update is shown in Figure 3.4. The detailed algorithm is shown in Algorithm 13.

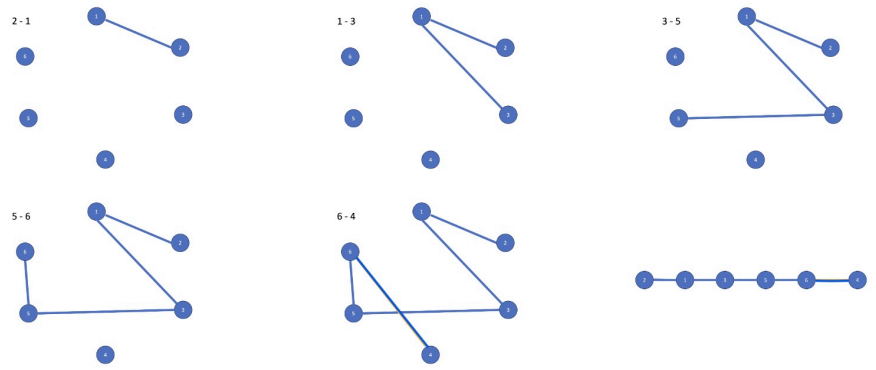


Figure 3.4: Chain network illustration.

Algorithm 13 Chain Update

procedure ChainUpdate(T, n_sites, n, $\boldsymbol{\beta}$)
    Input: Total epochs T, number of sites n_sites, number of selected sites n
        Parameter matrix $\boldsymbol{\beta}$ with vectors for each site
    Output: Updated parameter matrix $\boldsymbol{\beta}$
    Initialize index list: $ind \leftarrow \text{list}(range(n\_sites))$
    for $t = 1$ to T **do**
        Shuffle index list for random chain: random.shuffle(ind)
        for $i = 1$ to $n - 1$ **do**
            Call MKT for model transfer between vectors: MKT($\boldsymbol{\beta}[ind[i-1]], \boldsymbol{\beta}[ind[i]]$)
        end for
    end for
    return $\boldsymbol{\beta}$
end procedure

3.5.3 Star Network

Lastly, for Star, it is a centralized updating algorithm. Each epoch, there is a node that acts as the center of all nodes. This node is always the sender of information,

and other nodes act as receivers sequentially. For each iteration, the center node receives information from one other node and updates its parameters. A plot of the Star update is shown in 3.5. The detailed algorithm is shown in Algorithm 14.

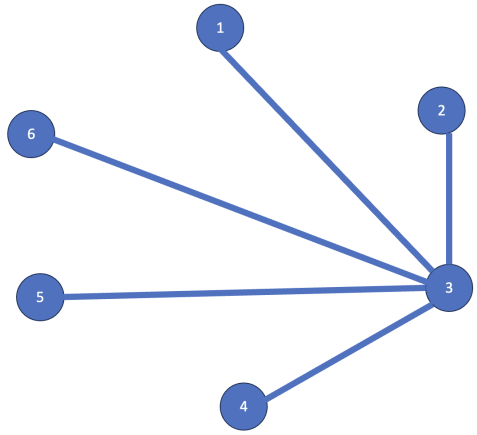


Figure 3.5: Star network illustration.

Algorithm 14 Star Update

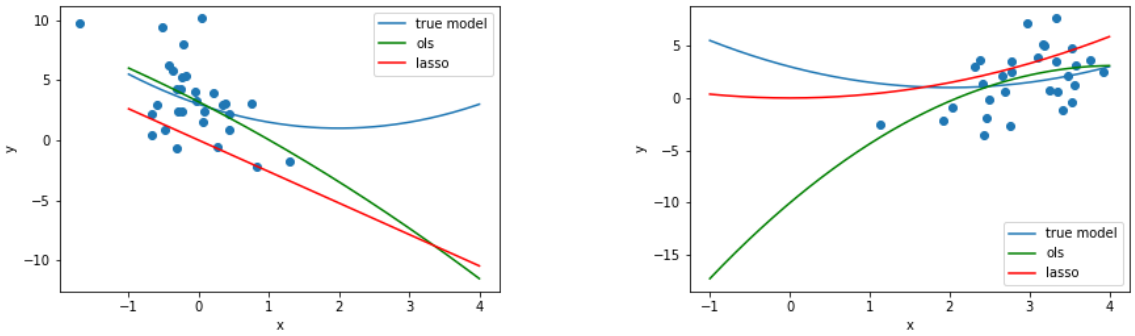
procedure StarUpdate(T, n_sites, $\boldsymbol{\beta}$)
    Input: Total epochs T, number of sites n_sites
        Parameter matrix $\boldsymbol{\beta}$ with vectors for each site
    Output: Updated parameter matrix $\boldsymbol{\beta}$
    for $t = 1$ to T **do**
        Initialize index list: $ind \leftarrow \text{list}(range(n\_sites))$
        Shuffle index list to randomize central node: random.shuffle(ind)
        Set central node (sender): $central \leftarrow ind[0]$
        for $i = 1$ to $n\_sites - 1$ **do**
            Receive updates from central node: MKT($\boldsymbol{\beta}[central], \boldsymbol{\beta}[ind[i]]$)
        end for
    end for
    return $\boldsymbol{\beta}$
end procedure

3.6 Toy Example

In this simulation, we aim to see how each method performs when we have a regression problem and each site gets partial information of the regression. For this setup, the true model is a quadratic model $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$. We set $\beta_0 = 3$, $\beta_1 = -2$, $\beta_2 = 4$ and $\epsilon \sim N(0, 3)$. There are two sites, each with 30 rows: the X value of site 1 is generated from Normal$(0, 0.5)$; the X value of site 2 is generated from Normal$(3, 0.5)$.

Figure 3.6 shows the data for each site, as well as the fitted curves from Ordinary Least Squares (OLS) and LASSO regression. From the plots, OLS fails to capture the real trend in the dataset, whereas LASSO corrects it by some amount. The true model is a quadratic curve that concaves up. For Site 1, both OLS and LASSO predict a concave down pattern. For Site 2, OLS is concave down but LASSO corrects the concavity.

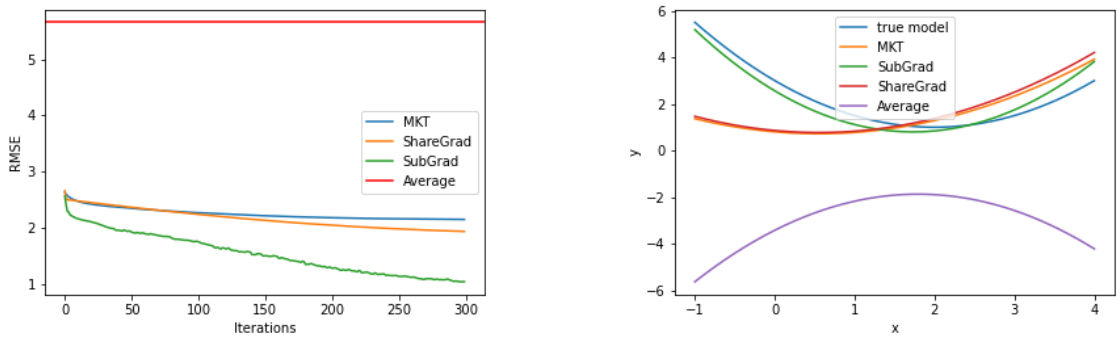
From panel (a) of 3.7, Subgradient Descent (Fusion method) has the lowest RMSE. MKT and Shared Gradient methods have similar performance, but still significantly better than taking the average of the two OLS models. Moving on to panel (b), almost all methods successfully predict the concavity of the curve except for the simple average. At the same time, the curve of Subgradient Descent is the closest to the true model.



(a) Fitted curves for site 1

(b) Fitted curves for site 2

Figure 3.6: Data and fitted curves for each site.



(a) RMSE results

(b) Fitted curves

Figure 3.7: RMSE and fitted curves for different methods.

3.7 Application on MNIST Data

The MNIST dataset, a cornerstone in Machine Learning, is a large database of handwritten digits used extensively for training and testing image processing systems. It consists of 60000 training images and 10000 test images, each a 28 by 28 gray-scale depiction of digits 0 through 9. This dataset has become a benchmark for evaluating Machine Learning algorithms, providing a straightforward challenge for methods in image classification. For example, it is often the first dataset used to demonstrate the effectiveness of techniques ranging from simple linear classifiers

to complex Deep Learning models. Its simplicity and comprehensive coverage of variations in handwriting make it an ideal starting point for experimenting with new algorithms and teaching Machine Learning concepts. In this section, MKT is defined as updating the receiver only, unless specified otherwise.

3.7.1 Binary Classification with Distribution Shift

We consider a binary classification problem. Specifically, we only selected 3s and 5s from the MNIST dataset. We created two sites, with the first site containing 200 rows with 95 percent 3s and 5 percent 5s, and the other site containing 100 rows with 2 percent 3s and 98 percent 5s.

For the testing dataset, we have perfectly balanced data with equally distributed 3s and 5s. The goal is to see how information sharing will affect the testing performance of individual models.

Figure 3.8 shows the change in accuracy for this simulation across different methods. From the plot, we can see that the average model produces an accuracy of around 90 percent, which is already significantly higher than local estimation, which is near 65 percent for each site. On the other hand, Gradient Sharing, Subgradient Descent, and the MKT method outperform the simple average, with Subgradient Descent achieving the highest accuracy among all methods.

3.7.2 Classification with Partial Information

For this simulation, we consider multiple digits from the dataset. Specifically, we let the first dataset contain digits 0 and 4, while the second contains 0 and 8. The reason for doing so is that we want to see whether information can be transferred when part of the data is missing. In particular, site 1 has 200 rows, with 120 of 0s

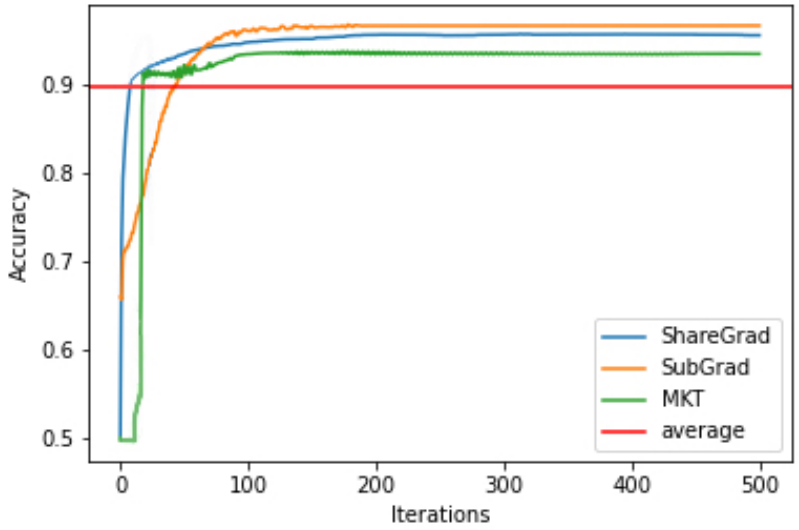


Figure 3.8: Accuracy for two sites with distribution shift.

and 80 of 4s, whereas site 2 has 100 rows, with 60 of 0s and 40 of 8s. The testing dataset has around 3000 rows with a mixture of 0s, 4s, and 8s.

Figure 3.9 shows the change in accuracy for this simulation across different methods. For local estimation, each site has an accuracy of around 65 percent. This is reasonable because the testing data is very balanced and each site by itself only contains 2 out of the 3 digits. By simply taking the average of the models, predicting accuracy is not improved by a significant amount. However, MKT, Sharing Gradient, and Subgradient Descent can all bump up the prediction accuracy, with Subgradient Descent improving the accuracy to nearly 90 percent.

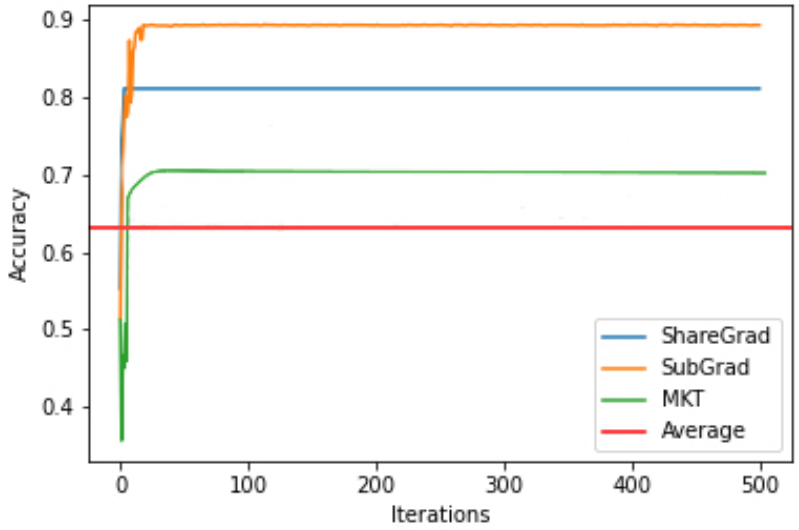


Figure 3.9: Accuracy for two sites with partial information.

3.7.3 Classification with Multiple Sites

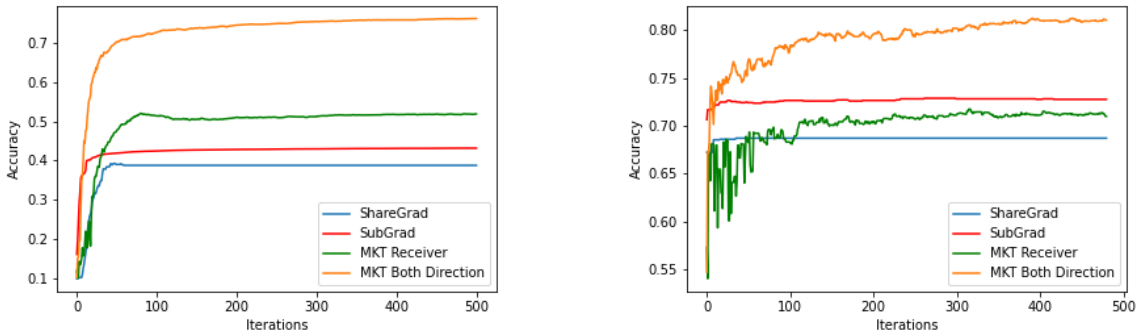
10 Sites

For this simulation, we have 10 sites in total. Each site has 100 rows of data, with one of the digits being dominant. Precisely, the dominant digit has 82 rows, and the rest of the digits have 2 rows each. For the 10 sites case, sites 1 to 9 have their site number as the dominant digit, and site 10 has 0 as its dominant digit. For testing data, the first case involves a balanced dataset of 5000 rows, with every digit evenly distributed. This corresponds to consensus testing, which means each local dataset contains part of the information in the testing data, and they learn from each other to get the missing pieces. The second case involves testing data with dominant 8s. However, 8 constitutes 60 percent of all the data, and the rest of the digits are about 4 percent each. This corresponds to local testing, which means the local data is representative of the population, but because of limited samples, we

need to leverage information from other sites to adjust the local model.

For estimation with local data only, when the testing data is balanced, each site has a prediction accuracy of less than 10 percent. Averaging the models from each site does not really help with prediction accuracy. From Figure 3.10 panel (a), we can see that all these methods can perform significantly better than simply taking an average. Among these methods, Sharing Gradient and Fusion Method can boost performance to nearly 40 percent, whereas MKT with receiver only can increase it to 50 percent. When updating the sender at the same time, MKT can achieve an accuracy as high as about 75 percent.

When the testing data has a local pattern, we only look at the testing accuracy of the 8th site. The pattern of different methods is very similar, but the difference in accuracy is smaller than in the previous setup. Almost all methods can boost the accuracy to over 70 percent, with MKT Both Directions being able to achieve an accuracy of over 80 percent.



(a) Consensus testing data

(b) Local testing data

Figure 3.10: Accuracy for 10 sites with dominant digit.

100 Sites

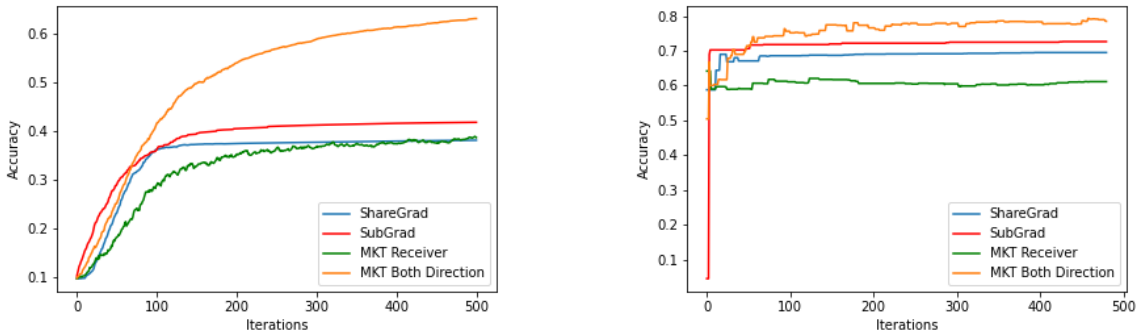
For this simulation, we have 100 sites in total. Each site has 100 rows of data, with one of the digits being dominant. Precisely, the dominant digit has 82 rows and the rest of the digits have 2 rows each. For this case, sites 1 to 9 have their site number as the dominant digit, and site 10 has 0 as its dominant digit. For sites from 11 to 100, the remainder of dividing by 10 is the dominant digit. For testing data, the first case is we have a balanced dataset of 5000 rows, and every digit is evenly distributed. The second case is we have biased testing data with dominant 8s. Similar to the last experiment, 8 constitutes only 60 percent of all the data and the rest of the digits are about 4 percent each.

For estimation with local data only, when the testing data is balanced, each site has a prediction accuracy of less than 10 percent. Averaging the models from each site does not really help with prediction accuracy. From Figure 3.11 panel (a), we can see that all these methods can do significantly better than simply taking an average. The major difference between 100 sites and 10 sites is that convergence becomes slower when there are more sites, as it takes more iterations for each site to be updated.

When the testing data is the second case, which is dominated by 8s, we only look at the testing accuracy of the 8th, 18th, 28th, ..., 98th sites. The pattern is very similar to that of the 10 sites.

3.7.4 Network Efficiency

In this simulation study, we examine network efficiency by monitoring convergence speed under different scenarios. We focus on the MKT algorithm, with an assumption of a fully connected network.



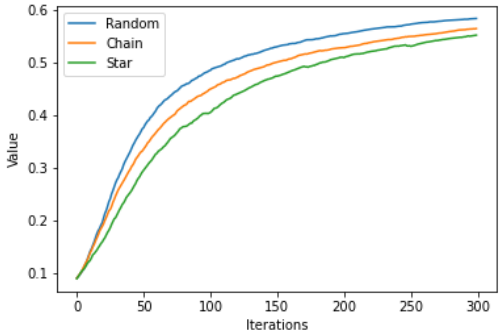
(a) Consensus testing data

(b) Local testing data

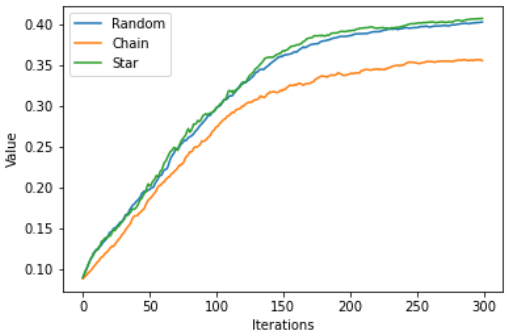
Figure 3.11: Accuracy for 100 sites with one dominant digit.

We explore three different ways of communication. The first one is Random communication. For each epoch, we randomly select 9 pairs of sites to update, with one of them being the sender and the other one being the receiver. The second method is through a Chain. We randomly select 10 sites and connect them into a Chain. The updating order is determined by the Chain that is randomly generated in each iteration. For each epoch, information sharing always starts from the head of the Chain and gradually propagates to the tail of it. The last one is Star. There is a node that acts as the center of all nodes. This node is always the sender of information, and other nodes act as receivers sequentially. For each iteration, the center node receives information from one other node and updates its parameters.

We explore these three network structures on two different setups. One is updating both senders and receivers, and the second is updating receivers only. From Figure 3.12, from panel (a), we can see that Random connection has the best convergence speed and accuracy, and Chain and Star are the next ones. From panel (b), when updating receivers only, Random and Star have the same convergence speed and accuracy, whereas Chain is slightly worse.



(a) MKT with bidirectional update



(b) MKT with receiver update

Figure 3.12: Network efficiency for 100 sites with one dominant digit.

3.8 Theoretical Results

Consider the setup with two sites, each site running a logistic regression problem. Define $\boldsymbol{\beta_1^t}$ and $\boldsymbol{\beta_2^t}$ as the coefficients of site 1 and site 2 at the t-th iteration. Consider the loss function:

$$L(\boldsymbol{\beta_1^t}, \mathcal{B}_1, \boldsymbol{p}_2) = L_C(\boldsymbol{p}_1, \boldsymbol{y}_2) + D_{KL}(\boldsymbol{p}_2||\boldsymbol{p}_1), \tag{3.12}$$

where:

$$\begin{aligned}
&\boldsymbol{p}_1 \in \mathbb{R}^n, \{p_{1,i}\} = p(\boldsymbol{y}_{1,i} = 1|\boldsymbol{\beta_1^t}),\\
&\boldsymbol{p}_2 \in \mathbb{R}^n, \{p_{2,i}\} = p(\boldsymbol{y}_{1,i} = 1|\boldsymbol{\beta_2^t}),\\
&L_C(\boldsymbol{p}_1, \boldsymbol{y}_1) = -\sum_{i=1}^{n} h(y_{1,i}) \cdot p_{1,i},\\
&D_{KL}(\boldsymbol{p}_2||\boldsymbol{p}_1) = \sum_{i=1}^{n} \left((1 - p_{2,i}) \log \frac{1 - p_{2,i}}{1 - p_{1,i}} + p_{2,i} \log \frac{p_{2,i}}{p_{1,i}} \right).
\end{aligned}$$

Without loss of generality, we prove the convergence of $\boldsymbol{\beta_1^t}$ under the Mutual Knowledge Transfer framework when site 1 is the receiver.

Lemma 3.8.1. *Given data* $\mathcal{B}_1 = \{\boldsymbol{X_1}, \boldsymbol{y_1}\}$, *let* $L(\boldsymbol{\beta_1^t}) = L(\boldsymbol{\beta_1^t}, \mathcal{B}_1, \boldsymbol{p}_1)$, *then* $\|\frac{dL(\boldsymbol{\beta_1^t})}{d\boldsymbol{\beta_1}}\|^2$ *is bounded.*

Proof. Given $y_{1,i} \in \{0, 1\}\ \forall i$:

$$\begin{aligned} p_{1,i} &= \frac{1}{1 + \exp(-\boldsymbol{\beta}_1^T x_i)}, \\ \frac{dL(\boldsymbol{\beta_1^t})}{d\boldsymbol{\beta_1^t}} &= -[(\boldsymbol{y_1} - \boldsymbol{p_1}) + (\boldsymbol{p_1} - \boldsymbol{p_2})^T]\boldsymbol{X_2} \\ &= (2\boldsymbol{p_1} - \boldsymbol{p_2} - \boldsymbol{y_1})^T \boldsymbol{X_1}. \end{aligned}$$

Since $p_{1,i}, p_{2,i} \in [0, 1]\ \forall i$,

$$2p_{1,i} - p_{2,i} - y_{1,i} \in [-2, 2]\ \forall i.$$

Therefore,

$$\|\frac{dL(\boldsymbol{\beta_1^t})}{d\boldsymbol{\beta_1^t}}\|^2 \leq 4\sum_{i,j} x_{ij}^2.$$

□

Theorem 3.8.2. *Let* $R, G > 0$. *Let* T *denotes number of iterations of Stochastic Gradient Descent. Suppose* $\boldsymbol{\beta_1^*}$ *(optimum) exists. We use* dL *to denote* $\frac{dL((\boldsymbol{\beta_1^t}))}{d\boldsymbol{\beta_1^t}}$ *and* $\tilde{dL}$ *to denote the direction of update for a random batch. Assume* $E(\|\boldsymbol{\beta_1^1} - \boldsymbol{\beta_1^*}\|^2) \leq R^2$, $E(\tilde{dL}) = dL$ *and* $E(\|\tilde{dL}\|^2) \leq G^2$. *Consider the iterates of the Mutual Knowledge Transfer method:*

$$\boldsymbol{\beta_1^{t+1}} = \boldsymbol{\beta_1^t} - \eta\tilde{dL}. \tag{3.13}$$

The iterates satisfy

$$E\left[L\left(\frac{1}{T}\sum_{t=1}^{T}\boldsymbol{\beta_1^t}\right) - L(\boldsymbol{\beta_1^*})\right] \leq \frac{RG}{\sqrt{T}}. \tag{3.14}$$

Proof. Since $L_C(\boldsymbol{p_1})$ is convex w.r.t. $\boldsymbol{\beta_1}$, $D_{KL}(\boldsymbol{p_2}||\boldsymbol{p_1})$ is convex, then we can say $L(\boldsymbol{\beta_1}) = L_C(\boldsymbol{p_1}) + D_{KL}(\boldsymbol{p_2}||\boldsymbol{p_1})$ is convex.

$$\begin{aligned}
E\left[\|\boldsymbol{\beta_1^{t+1}} - \boldsymbol{\beta_1^*}\|_2^2 \mid \boldsymbol{\beta_1^t}\right] &= E\left[\|\boldsymbol{\beta_1^t} - \eta\tilde{dL} - \boldsymbol{\beta_1^*}\|_2^2 \mid \boldsymbol{\beta_1^t}\right] \\
&= E\left[\|\boldsymbol{\beta_1^t} - \boldsymbol{\beta_1^*}\|_2^2 - 2\eta\tilde{dL}^T\left(\boldsymbol{\beta_1^t} - \boldsymbol{\beta_1^*}\right) + \eta^2\|\tilde{dL}\|_2^2 \mid \boldsymbol{\beta_1^t}\right] \\
&\leq \|\boldsymbol{\beta_1^t} - \boldsymbol{\beta_1^*}\|^2 - 2\eta\left[L(\boldsymbol{\beta_1^t}) - L(\boldsymbol{\beta_1^*})\right] + \eta^2 E\left[\|\tilde{dL}\|_2^2 \mid \boldsymbol{\beta_1^t}\right].
\end{aligned}$$

By law of total expectation,

$$\begin{aligned}
0 \leq E\left[E\left[\|\boldsymbol{\beta_1^{t+1}} - \boldsymbol{\beta_1^*}\|_2^2 \mid \boldsymbol{\beta_1^t}\right]\right] &= E\left[\|\boldsymbol{\beta_1^{t+1}} - \boldsymbol{\beta_1^*}\|_2^2\right] \\
&\leq E\left[\|\boldsymbol{\beta_1^t} - \boldsymbol{\beta_1^*}\|^2\right] - 2\eta\left[E\left(L(\boldsymbol{\beta_1^t})\right) - L(\boldsymbol{\beta_1^*})\right] + \eta^2 E\left[\|\tilde{dL}\|_2^2\right] \\
&\leq E\left[\|\boldsymbol{\beta_1^1} - \boldsymbol{\beta_1^*}\|^2\right] - 2\eta\sum_{k=1}^{T}\left[E\left(L(\boldsymbol{\beta_1^k})\right) - L(\boldsymbol{\beta_1^*})\right] + \eta^2 tG^2.
\end{aligned}$$

Rearranging the terms:

$$2\eta\sum_{k=1}^{t}\left[E\left(L(\boldsymbol{\beta_1^k})\right) - L(\boldsymbol{\beta_1^*})\right] \leq E\left[\|\boldsymbol{\beta_1^1} - \boldsymbol{\beta_1^*}\|^2\right] + T\eta^2 G^2.$$

Since $E\left[\|\boldsymbol{\beta_1^1} - \boldsymbol{\beta_1^*}\|^2\right] = R^2$, dividing both sides by $2\eta T$ we have

$$E\left[L\left(\frac{1}{T}\sum_{k=1}^{T}\boldsymbol{\beta_1^k}\right) - L\left(\boldsymbol{\beta_1^*}\right)\right] \leq \frac{R^2}{2\eta T} + \frac{G^2\eta}{2}.$$

We want to minimize RHS w.r.t η

$$\frac{d}{d\eta}\left(\frac{R^2}{2\eta T} + \frac{G^2\eta}{2}\right) = -\frac{R^2}{2\eta^2 T} + \frac{G^2}{2}.$$

Set derivative to 0 and solve for η, we have

$$\begin{aligned}
0 &= -\frac{R^2}{2\eta^2 T} + \frac{G^2}{2} \\
\frac{R^2}{2\eta^2 T} &= \frac{G^2}{2} \\
\eta^2 &= \frac{R^2}{TG^2} \\
\hat{\eta} &= \frac{R}{\sqrt{T}G}.
\end{aligned}$$

Plug it back into the original solution:

$$\begin{aligned}
E\left[L\left(\frac{1}{T}\sum_{k=1}^{T}\boldsymbol{\beta_1^k}\right) - L\left(\boldsymbol{\beta_1^*}\right)\right] &\leq \frac{R^2}{2\frac{R}{\sqrt{T}G}T} + \frac{G^2\frac{R}{\sqrt{T}G}}{2} \\
&= \frac{R^2}{\frac{2R\sqrt{T}}{G}} + \frac{GR}{2\sqrt{T}} \\
&= \frac{GR}{2\sqrt{T}} + \frac{GR}{2\sqrt{T}} \\
&= \frac{GR}{\sqrt{T}}.
\end{aligned}$$

□

3.9 Discussion

Decentralized Federated Learning has gradually become a popular topic as privacy issues become more prominent in Machine Learning. In Decentralized Federated Learning problems, different sites transfer information to each other, without the need for a central server. Two of the major questions are what to share and how to share. In this project, we explore different ways of sharing information between sites and compare their respective performances. Starting from local estimation only, we explore the methods of data aggregation, sharing gradients, sharing model parameters, and Mutual Knowledge Transfer. We explore the performances of each method under different scenarios, such as distribution shifts, partial information, as well as consensus testing and local testing cases. We conclude that sharing models typically work the best, as the Fusion method and MKT normally outperform other methods. Additionally, we explore different transmission methods including Random, Chain, and Star. We show that using the same model, different transmission topologies can lead to different efficiencies.

Bibliography

M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

S. A. Aketi, A. Hashemi, and K. Roy. Global update tracking: A decentralized learning algorithm for heterogeneous data. *ArXiv*, 2023.

H. Allende-Cid, C. Moraga, R. Monge, and H. Allende. The problem of centralizing distributed data sources in the regression task. *Consensus*, 2016.

M. Assran, N. Loizou, N. Ballas, and M. Rabbat. Stochastic gradient push for distributed deep learning. In *International Conference on Machine Learning*, pages 344–353. PMLR, 2019.

T. C. Aysal, M. E. Yildiz, A. D. Sarwate, and A. Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal Processing*, 57(7):2748–2761, Feb 2009.

E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov. How to backdoor federated learning. In *International Conference on Artificial Intelligence and Statistics*, pages 2938–2948. PMLR, 2020.

J. Y. Baladi and J. L. Hendrix. Expert-based single point failure analysis. *INCOSE International Symposium*, 21, 2011. doi: 10.1002/j.2334-5837.2011.tb01212.x.

A. Barbero and S. Sra. Modular proximal optimization for multidimensional total-variation regularization. *arXiv preprint*, 2014. doi: arXiv:1411.0589.

L. Baruh, E. Secinti, and Z. Cemalcılar. Online privacy concerns and privacy management: A meta-analytical review. *Journal of Communication*, 67:26–53, 2017. doi: 10.1111/JCOM.12276.

S. Boyd. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2010. doi: 10.1561/2200000016. URL `https://doi.org/10.1561/2200000016`.

S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: Design, analysis, and applications. In *Proceedings IEEE INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1653–1664. IEEE, 2005.

S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, Jun 2006.

S. Chen, D. Yu, Y. Zou, J. Yu, and X. Cheng. Decentralized wireless federated learning with differential privacy. *IEEE Transactions on Industrial Informatics*, 18(9):6273–6282, Jan 2022.

Y. Chen, Y. Ning, and H. Rangwala. Asynchronous online federated learning for edge devices. *ArXiv*, 2019.

H. Cho, D. J. Wu, and B. Berger. Secure genome-wide association analysis using multiparty computation. *Nature Biotechnology*, 36:547, 2018.

L. Condat. A direct algorithm for 1d total variation denoising. *HAL preprint*, 2012.

L. Corinzia and J. Buhmann. Variational federated multi-task learning. *ArXiv*, 2019.

P. L. Davies and A. Kovac. Local extremes, runs, strings and multiresolution. *Annals of Statistics*, 29(1):1–65, 2001.

A. Durrant, M. Markovic, D. Matthews, D. May, J. Enright, and G. Leontidis. The role of cross-silo federated learning in facilitating data sharing in the agri-food sector. *Computers and Electronics in Agriculture*, 193:106648, Feb. 2022. ISSN 0168-1699. doi: 10.1016/j.compag.2021.106648. URL `http://dx.doi.org/10.1016/j.compag.2021.106648`.

C. Dwork. Differential privacy. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Automata, Languages and Programming. ICALP 2006. Lecture Notes in Computer Science*, Lecture Notes in Computer Science. Springer, 2006.

C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Theory of Cryptography Conference*, pages 265–284, 2006.

T. El-Sayed, A. Moustafa, M. Elrashidy, and A. El-Sayed. Optimizing federated learning approach: Literature survey and open points. In *2023 3rd International Conference on Electronic Engineering (ICEEM)*, pages 1–5. IEEE, 2023.

J. Fan and R. Li. Variable selection via nonconcave penalized likelihood and its

oracle properties. *Journal of the American Statistical Association*, 96(456):1348–1360, 2001.

M. Ferrag, O. Friha, L. Maglaras, H. Janicke, and L. Shu. Federated learning for connected and automated vehicles: A survey of existing approaches and challenges. *IEEE Access*, 9:143634–143655, 2021.

C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage. Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*, 2018.

H. Hoefling. A path algorithm for the fused lasso signal approximator. *Journal of Computational and Graphical Statistics*, 19(4):984–1006, 2010.

L. Jena, N. K. Kamila, and S. Mishra. Privacy preserving distributed data mining with evolutionary computing. pages 259–267, 2013. doi: 10.1007/978-3-319-02931-3_29.

S. Jha, L. Kruger, and P. McDaniel. Privacy preserving clustering. In *European Symposium on Research in Computer Security*, pages 397–417. Springer, 2005.

J. Jiang and L. Hu. Decentralised federated learning with adaptive partial gradient aggregation. *CAAI Trans. Intell. Technol.*, 5:230–236, 2020.

N. Johnson. A dynamic programming algorithm for the fused lasso and l_0-segmentation. *Journal of Computational and Graphical Statistics*, 22(2):246–260, 2013.

G. A. Kaissis, M. R. Makowski, D. Rückert, and R. F. Braren. Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2(6):305–311, 2020. doi: 10.1038/s42256-020-0186-1.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

V. Kolmogorov, T. Pock, and M. Rolinek. Total variation on a tree. *SIAM Journal of Imaging Sciences*, 9(2):605–636, 2016.

J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon. Federated learning: Strategies for improving communication efficiency, 2016.

K.Tam, L.Li, B.Han, C.Xu, and H.Fu. Federated noisy client learning. *IEEE Transactions on Neural Networks and Learning Systems*, PP, 2023. doi: 10.1109/TNNLS.2023.3336050. Epub ahead of print.

A. Lalitha, S. Shekhar, T. Javidi, and F. Koushanfar. Fully decentralized federated learning. In *Third Workshop on Bayesian Deep Learning (NeurIPS)*, 2018.

L. Landrieu and G. Obozinski. Cut pursuit: fast algorithms to learn piecewise constant functions on general weighted graphs. *HAL preprint*, 2015.

J. Le, X. Lei, N. Mu, H. Zhang, K. Zeng, and X. Liao. Federated continuous learning with broad network architecture. *IEEE Transactions on Cybernetics*, 2021.

C. Li and H. Li. Network-constrained regularization and variable selection for analysis of genomic data. *Bioinformatics*, 24(9):1175–1182, 2008. doi: doi: 10.1093/bioinformatics/btn081.

C. Li, G. Li, and P. K. Varshney. Decentralized federated learning via mutual knowledge transfer. *IEEE Internet of Things Journal*, 2021. doi: 10.1109/JIOT.2021.3078543. Accepted for publication, content may change prior to final publication.

S. Li, Y. Cheng, W. Wang, Y. Liu, and T. Chen. Learning to detect malicious clients for robust federated learning. *ArXiv*, 2020.

S. Li, E. Ngai, F. Ye, and T. Voigt. Auto-weighted robust federated learning with corrupted data sources. *ACM Transactions on Intelligent Systems and Technology*, 13(5):1–20, June 2022. ISSN 2157-6912. doi: 10.1145/3517821. URL `http://dx.doi.org/10.1145/3517821`.

Z. Li and L. Chen. Communication-efficient decentralized zeroth-order method on heterogeneous data. In *2021 13th International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 2021.

K. G. Liakos, P. Busato, D. Moshou, S. Pearson, and D. Bochtis. Machine learning in agriculture: A review. *Sensors (Basel, Switzerland)*, 18, 2018. doi: 10.3390/s18082674.

F. Liu, M. Li, X. Liu, T. Xue, J. Ren, and C. Zhang. A review of federated meta-learning and its application in cyberspace security. *Electronics*, 12(15):3295, 2023.

A. Manglik, J. Singh, and M. K. Tiwari. A comprehensive review of the literature on machine learning-based road safety prediction techniques for internet of vehicles

(iov)-enabled vehicles. *Tuijin Jishu/Journal of Propulsion Technology*, 2023. doi: 10.52783/tjjpt.v44.i4.2030.

E. T. Martínez Beltrán et al. Decentralized federated learning: Fundamentals, state of the art, frameworks, trends, and challenges. *IEEE Communications Surveys & Tutorials*, 25(1):2983–3013, 2022.

B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Arcas. Communication-efficient learning of deep networks from decentralized data. In A. Singh and J. Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 20–22 Apr 2017. URL `http://proceedings.mlr.press/v54/mcmahan17a.html`.

A. Nedic. Asynchronous broadcast-based convex optimization over a network. *IEEE Transactions on Automatic Control*, 56(6):1337–1351, Sep 2010.

A. Nedić, A. Olshevsky, and M. G. Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *Proc. IEEE*, 106(5):953–976, Apr. 2018.

H. Nguyen, L. Kieu, T. Wen, and C. Cai. Deep learning methods in transportation domain: a review. *IET Intelligent Transport Systems*, 2018. doi: 10.1049/IET-ITS.2018.0064.

A. Nilsson, S. Smith, G. Ulm, E. Gustavsson, and M. Jirstrand. A performance evaluation of federated learning algorithms. In *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*, 2018.

C. Pandey, S. Sharma, and P. Matta. Data analysis and modeling of body sensor network in healthcare application. In *2022 6th International Conference on Electronics, Communication and Aerospace Technology*, pages 590–596, 2022. doi: 10.1109/ICECA55336.2022.10009487.

M. Panigrahi, S. Bharti, and A. Sharma. A review on client selection models in federated learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(1), 2023.

C. Pappas, D. Chatzopoulos, S. Lalis, and M. Vavalis. Ipls: A framework for decentralized federated learning. In *2021 IFIP Networking Conference (IFIP Networking)*, pages 1–6. IEEE, 2021.

V. Sadhanala, Y. Wang, and R. Tibshirani. Graph sparsification approaches for laplacian smoothing. *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, 51:1250–1259, 2016.

M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

Y. M. Saputra, D. T. Hoang, D. N. Nguyen, E. Dutkiewicz, M. D. Mueck, and S. Srikanteswara. Energy demand prediction with federated learning for electric vehicle networks, 2019.

F. Sattler, S. Wiedemann, K.-R. Müller, and W. Samek. Robust and communication-efficient federated learning from non-iid data. *IEEE Transactions on Neural Networks and Learning Systems*, 31(9):3400–3413, Nov 2019.

G. Schumann, J.-P. Awick, and J. Marx Gómez. Natural language processing using federated learning: A structured literature review. In *2023 IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*, pages 1–7. IEEE, 2023.

M. J. Sheller, G. A. Reina, B. Edwards, J. Martin, and S. Bakas. Multi-institutional deep learning modeling without sharing patient data: A feasibility study on brain tumor segmentation. In *International MICCAI Brainlesion Workshop*, pages 92–104. Springer, 2019.

Y. Shi, Y. Zhou, and Y. Shi. Over-the-air decentralized federated learning. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 455–460. IEEE, 2021.

A. J. Smola and R. Kondor. Kernels and regularization on graphs. In B. Schölkopf and M. Warmuth, editors, *Learning Theory and Kernel Machines*, volume 2777 of *Lecture Notes in Computer Science*, pages 144–158. Springer, Berlin, 2003.

H. Sun and H. Li. A bayesian approach for graph-constrained estimation for high-dimensional regression. *Int J Syst Synth Biol*, 1(2):255–272, 2010.

J.-x. Sun. Complexity of software system and optimal design for failure management. *Computer and Modernization*, 2009.

M. Talistu, T.-S. Moh, and M. Moh. Gossip-based spectral clustering of distributed data streams. In *2015 International Conference on High Performance Computing & Simulation (HPCS)*, 2015.

M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural

networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

W. Tansey and J. Scott. A fast and flexible algorithm for the graph-fused lasso. *arXiv preprint*, 2015. doi: arXiv:1505.06475.

R. Tibshirani and J. Taylor. The solution path of the generalized lasso. *Annuls of Statistics*, 39(3):1335–1371, 2011.

R. Tibshirani, M. Saunders, S. Rosset, J. Zhu, and K. Knight. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B*, 67(1): 91–108, 2005.

R. Tibshriani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, pages 267–288, 1996.

V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu. Data poisoning attacks against federated learning systems. In *European Symposium on Research in Computer Security*, pages 480–501. Springer, 2020.

Y. Tsarenko and D. Tojib. Examining customer privacy concerns in dealings with financial institutions. *Journal of Consumer Marketing*, 26:468–476, 2009. doi: 10.1108/07363760911001529.

R. Turn, N. Shapiro, and M. Juncosa. Privacy and security in centralized vs. decentralized databank systems. *Policy Sciences*, 7:17–29, 1976. doi: 10.1007/BF00146019.

Y.-X. Wang, J. Sharpnack, A. Smola, and R. J. Tibshirani. Trend filtering on graphs, 2016a.

Y.-X. Wang, J. Sharpnack, A. J. Smola, and R. J. Tibshirani. Trend filtering on graphs. *Journal of Machine Learning Research*, 17(105):1–41, 2016b. URL `http://jmlr.org/papers/v17/15-147.html`.

Z. Wang, R. Gupta, K. Han, H. Wang, A. Ganlath, N. Ammar, and P. Tiwari. Mobility digital twin: Concept, architecture, case study, and future challenges. *IEEE Internet of Things Journal*, 9(18):17452–17467, 2022. doi: 10.1109/JIOT.2022.3156028.

J. Wicaksana, Z. Yan, D. Zhang, X. Huang, H. Wu, X. Yang, and K.-T. Cheng. Fedmix: Mixed supervised federated learning for medical image segmentation. *IEEE Transactions on Medical Imaging*, 42:1955–1968, 2022. doi: 10.1109/TMI.2022.3233405.

S. Wu, D. Huang, and H. Wang. Network gradient descent algorithm for decentralized federated learning. *Journal of Business & Economic Statistics*, 41:806–818, 2022.

Y. Xia, D. Yang, W. Li, A. Myronenko, D. Xu, H. Obinata, H. Mori, P. An, S. Harmon, E. Turkbey, B. Turkbey, B. Wood, F. Patella, E. Stellato, G. Carrafiello, A. Ierardi, A. Yuille, and H. Roth. Auto-fedavg: Learnable federated averaging for multi-institutional medical image segmentation, 2021.

R. Xin, S. Kar, and U. Khan. Decentralized stochastic optimization and machine learning: A unified variance-reduction framework for robust performance and fast convergence. *IEEE Signal Processing Magazine*, 2020.

L. Xu, C. Jiang, Y. Qian, and Y. Ren. Privacy-accuracy trade-off in distributed data mining. pages 151–177, 2018. doi: 10.1007/978-3-319-77965-2_6.

H. Ye, L. Liang, and G. Li. Decentralized federated learning with unreliable communications. *IEEE Journal of Selected Topics in Signal Processing*, 16:487–500, 2021.

F. Zerka et al. Systematic review of privacy-preserving distributed machine learning from federated databases in health care. *JCO Clinical Cancer Informatics*, 2020.

W. Zhao, A. Bhushan, A. D. Santamaria, M. Simon, and C. Davis. Machine learning: A crucial tool for sensor design. *Algorithms*, 1(2):130–152, 2008. doi: 10.3390/a1020130.

H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2): 301–320, 2005.