An Implementation of General Fused Lasso

Based on - Graph -Structured Multi-task Regression and Efficient Optimization Method for General Fused Lasso;
Authors - Xi Chen, Seyoung Kim, Qihang Lin, Jaime G. Carbonell, Eric P. Xing
Source - [v1] Thu, 20 May 2010 on arXiv:1005.3579 https://arxiv.org/abs/1005.3579

Editor: Abhilash Chaudhary

Abstract

This project work aims to implement a **structured regularized regression** approach called graph-guided fused lasso (GFLasso) to **solve sparse multi-task learning problems**. The project implements an efficient algorithm called Proximal Gradient method required to solve the GFLasso optimization. The GFLasso regularization approach and the corresponding Proximal Gradient optimization algorithm is implemented on synthetically generated data. We then reported our results based on our implementation.

1. Introduction

Multi-task learning is a technique of learning multiple tasks **jointly** by analyzing data from **all** the tasks at the **same** time instead of analyzing it individually per task. This is a very helpful technique especially in cases when data is scarce – as we can borrow data from other related tasks to learn each task more effectively.

In general, while solving a lot of multi-task learning problems, tasks are assumed to be equally related to inputs without any structure and in all such cases, a mixed-norm regularization such as ℓ_1/ℓ_2 and ℓ_1/ℓ_∞[Appendix A3] multi-task regression has been used to find inputs relevant to all of the outputs jointly [1, 2]. But in many real-world multitask problems, the outputs are related in a more complex structure. In such cases, we need a new strategy that takes advantage of the structure of the output responses. This strategy is defined as follows -

Structured multi-task regression is a regression approach in which the output consists of multiple responses and the output structure is available in the form of a graph. These correlated response variables are dependent on a common set of inputs in a **sparse** but synergistic manner. Some common examples of such multi-task regression problems are

(a) In neuroscience application - to predict neural activities (outputs) in brain in response to words (inputs). Since **neural activities in the brain are locally correlated** in different brain regions rather than all regions sharing a similar response.[4].

(b) In stock prediction where some of the stock prices are more highly correlated than others [5].

In such a structured multi-task regression problem, the goal is to recover this structured sparsity pattern in the regression coefficients shared across all the correlated tasks that are related through a graph.

1.1 Motivation

This regression problem motivates the need to formulate a new strategy for structured multi-task regression that explicitly takes into account the complex dependency structure in the output variables represented as a graph and exploits the graph structure(or graph relation/connection) over

the output variables to estimate the regression coefficients. The present paper proposes one such technique called – Graph Guided Fused Lasso (GF Lasso).

2. Main Contributions

2.1 Graph-guided Fused Lasso for Sparse Structured Multitask Regression

GFLasso Formulation - Assume that the output structure of the K output variables is available as a graph G with a set of nodes $V = 1, ..., K$ and edges E. Such graphs are constructed by computing pairwise correlations based on y_k's and connecting two nodes with an edge if their correlation is above a given threshold ρ.

Here we use Pearson's correlation [Appendix A4] to define a correlation between out variables y_m and y_l as r_{ml} where r_{ml} denotes the weight of an edge $e(m, l)$ that represents the strength of correlation between the two nodes of the graph.

Given the graph G, if two output variables are connected with an edge in the graph, they tend to be influenced by the same set of covariates with similar strength. Also, the edge weights in the graph G contain information on how strongly the two output variables are related and thus share relevant covariates.

Standard Lasso- For a sample of N instances, each represented as a J dimensional input vector, and a K dimensional output vector, let $X = (x_1, x_2, ..., x_J)$ be a NxJ input matrix; $Y = (y_1, y_2, \dots y_K)$ be a NxK output matrix. For each of the K output variables, we have:

$$y_k = X\beta_k + \epsilon_k, \forall k = 1, ..., K \tag{1}$$

Where $\beta(k) = (\beta 1k, \beta 2k, \dots)$.T is a vector of regression coefficients for the kth output variable. And eta(k) is a vector of N independent zero mean Gaussian noise. If $B = (\beta_1, \beta_2, \dots \beta_K)$ denotes a JxK matrix of regression coefficients of ALL K response variables, then Lasso regression gives us –

$$\hat{B}^{lasso} = \text{argmin}\frac{1}{2}||Y - XB||_F^2 + \lambda||B||_1 \tag{2}$$

GFLasso employs an additional constraint over the standard lasso by fusing the $\beta_{jm}$and β_{jl}

$$\hat{B}^{GF} = \min_B f(B) = \frac{1}{2}||Y - XB||_F^2 + \lambda||B||_1 + \gamma \sum_{e=(m,l)\in E} \tau(r_{ml}) \sum_{j=1}^{J} |\beta_{jm} - sign(r_{ml})\beta_{jl}| \tag{3}$$

Where λ, γ are regularization parameters that control model complexity. Larger $\lambda \rightarrow$ greater fusion. τ_r weights the fusion penalty [Appendix A5]for each edge.$\beta_{jm}$and β_{jl} for highly correlated outputs with large r_{ml} receive greater fusion effect.

When this edge-level fusion penalty is applied to all the edges in the entire graph G in the GFLasso penalty, the overall effect is that each subset of output variables within a densely connected subgraph tends to have common relevant covariates.

2.2 Proximal-Gradient Method for Optimization

Although the optimization problem for GFLasso in Eq.1 is convex, it is not trivial to optimize it because of the non-smooth penalty function. In general, to solve fusion penalty, Quadratic Programming (QP) or Second-Order Cone-Programming (SOCP) techniques are used. But these techniques are very computational expensive and have huge convergence times. To overcome these challenges, the paper proposes a proximal-gradient method has a faster convergence rate and low computation complexity per iteration.

The "Proximal" approach - The "proximal" method optimizes a lower or upper bound of the original objective function, rather than optimizing the objective function directly. This lower or upper bound has a simpler form that allows for an easy optimization. Motivated by this idea, **Proximal-Gradient Method** technique follows the following steps –

(a) First reformulate the ℓ_1 and fusion penalty together into a max problem over auxiliary variables. Using the Reformulation of the non-smooth penalty term [Appendix A4], the overall penalty can be written as:

$$||BC||_1 \equiv \max_{\|A\|_\infty \leq 1} \langle A, BC \rangle \tag{4}$$

(b) After that, introduce its smooth lower bound and optimize that instead of optimizing the original penalty

$$f_\mu(B) = \max_{\|A\|_\infty \leq 1} \langle A, BC \rangle - \mu d(A) \tag{5}$$

(c) A Reformulation of the Non-smooth Penalty Term as in eq (2)

(d) Proximal Gradient Method

$$D = \max_{\|A\|_\infty \leq 1} d(A) = \frac{1}{2}||A||_F^2 = \frac{1}{2}J(K + |E|) \tag{6}$$

$$L = \lambda_{max}(X^T X) + L_\mu \leq \lambda_{max}(X^T X) + \frac{\lambda^2 + 2\gamma^2 max_{k \in V} d_k}{\mu} \equiv L_U \tag{7}$$

Algorithm 1 Proximal-Gradient Method for GFlasso

Input: $\mathbf{X}$, $\mathbf{Y}$, λ, γ, graph structure G, desired accuracy ϵ.

Initialization: Construct $C = (\lambda I, \gamma H)$; compute L_U according to (6) ; compute D in (5) and set $\mu = \frac{\epsilon}{2D}$; set $\mathbf{W}^0 = \mathbf{0} \in \mathbb{R}^{J \times K}$;

Iterate For $t = 0, 1, 2, \ldots$ until convergence of $\mathbf{B}^t$:

1. Compute $\nabla \widetilde{f}(\mathbf{W}^t)$ according to (11).
2. Perform the gradient descent step : $\mathbf{B}^t = \mathbf{W}^t - \frac{1}{L_U} \nabla \widetilde{f}(\mathbf{W}^t)$.
3. Set $\mathbf{Z}^t = -\frac{1}{L_U} \sum_{i=0}^{t} \frac{i+1}{2} \nabla \widetilde{f}(\mathbf{W}^i)$.
4. Set $\mathbf{W}^{t+1} = \frac{t+1}{t+3} \mathbf{B}^t + \frac{2}{t+3} \mathbf{Z}^t$.

Output: $\widehat{\mathbf{B}} = \mathbf{B}^t$

3. Experimental Setup

3.1 Dataset

We have implemented the algorithm on two synthetically generated datasets with sample size $N = 1000$, with 30 *features, i.e.* J and 40 *tasks,i.e.,* K. We have chosen 5 relevant features. The data split is $7:3$ ratio. The datasets generate their own ground truth coefficient matrix of size KxJ for the relevant features based on $y_k = X\beta_k + \epsilon_k \forall k = 1, \ldots K$ where $\beta_k = cos(c(1+x) + 3x)$ where x is a random value and $c \in [0, 2\pi]$. X is generated in two ways - a) randomly generated values of NxJ matrix, and, b) Gaussian X which generates a matrix of size NxJ where each value $\sim \mathcal{N}(0, 1)$.

For the comparison results and to see how does GFLasso perform relatively, we have used learning models *Lasso* and *MultiTaskLasso* libraries from sklearn to train on the same dataset and obtain regression coefficients which will be compared with the coefficients obtained from GFLasso algorithm.

We begin with initializing our graph G with identity matrix of size K and assign weight value of 1 to selected edges $e = (m, l) \in E$ to represent the connected nodes. This selection is done as every i_{th} node in $G_{i,j}$ is connected to $i_{(K/2)+1}$ node of $G_{i,j}$ and vice-a-versa. We started with $\lambda = 1$ and increased λ to several hundreds, however, with increasing λ, the algorithm cutoff would be sooner. Hence, we chose $\lambda = 2$ for our final algorithm. With similar cross validation we chose $\gamma = 1$ and $\epsilon = 1$. The value of $\mu = \frac{\epsilon}{2D}$ is decided as per original paper[1] where $D = \frac{1}{2}J(K + |E|)$ where D represents the abridgement between convex function and the lower bound of the convex function (J, K, E are number of features, tasks and edges respectively).

We begin with training the above dataset setup with 70% data for training and rest 30% for validation with $y_k = X\beta_k + \epsilon_k \forall k = 1, \ldots K$ and randomly generated X with $\rho = 0.5$, samples $N = 1000$, features $J = 30$ and tasks $K = 40$ and 5 relevant features. The algorithm then trains on *Lasso* and *MultiTaskLasso* of sklearn and our algorithm *GFLasso*. The following figure shows the receiver operating characteristic (ROC curve) for *Lasso, MultiTaskLasso and GFLasso* for the relevant features at index 0 and 2 and heatmap for all the relevant features. As can be seen from a) and b) that GFLasso has outperformed the other approaches with fewer false positives.

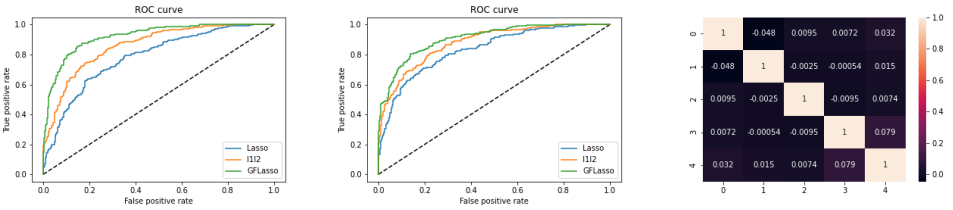

Figure 1: *from left* a) ROC curve for *Lasso, MultiTaskLasso and GFLasso* for the relevant feature at index 0 b) at index 2. c) Heatmap representing the correlation between relevant features

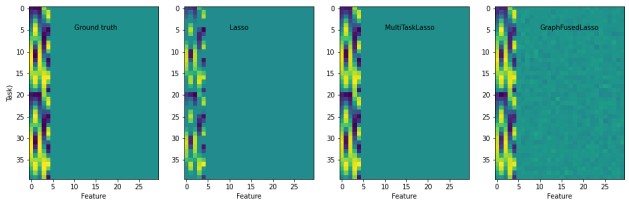

Figure 2: Regression Coefficients for *Ground Truth, Lasso, MultiTaskLasso and GFLasso*

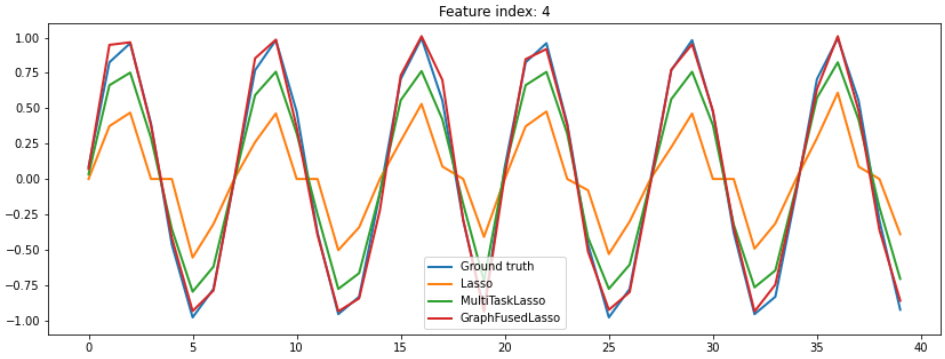

Figure 3: Regression coefficients for *Ground Truth, Lasso, MultiTaskLasso and GFLasso* of relevant feature at index 4

It can be inferred from Figure 4 that our GFLasso coefficients for relevant feature at index 4 are closer to the ground truth than lasso or $l1/l2$-regularized multi-task regression. As we move further in our analysis, we evaluate the performance of our GFLasso with proximal gradient function along with Stochastic Sub-Gradient method. Even though the paper originally compares the computation time with complex Second-Order Cone Programming(SOCP) and Quadratic Programming(QP) methods, we chose the above sub-gradient method because of the knowledge gap. First we change the value of J from 1000 to 10000 with step size of 1000 on both synthetic datasets. Then the value of N is changed from 500 to 10000 with step size of 1000 for both synthetic datsets. Similarly, we do this for vrying values of K from 1000 to 8000 with 1000 step size.

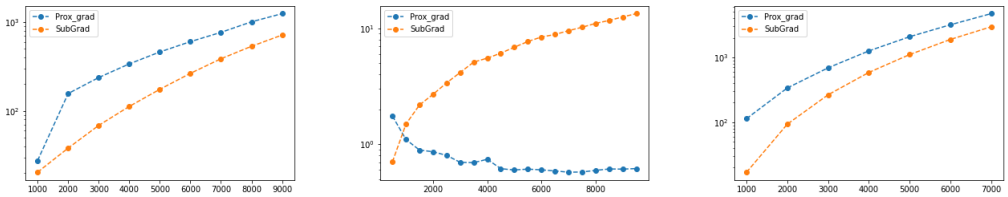

Figure 4: Computation Time by GFLasso with *Proximal Gradient, Stochastic Gradient Descent and Stochastic Sub-Gradient* for varying values of J, N, K *(from left respectively)* on normal distributed X.

4. Individual Contributions

Our study began with the research paper on GFLasso[7]. Nitin identified the problem statement and from there Andrew and Abhilash captured the theoretical concepts involved in the paper.
Further, Andrew started writing the report and Nitin along with Abhilash started implementing the GFLasso and Proximal Gradient in Python.
Later on Andrew and Abhilash played crucial role in generating the results and illustrations and Nitin followed up with the analysis.
Altogether, Abhilash, Andrew and Nitin played thorough roles in both theoretical and implementation aspects of the paper.

5. Conclusion

Learning and Future Directions The present work is one of the first to consider the graph structure (and hence the correlation) over the outputs in a multi-task problem. To do so , it proposes a new technique to solve such a structured multi-task regression problem and also proposes a new optimization method to solve such problems that is orders of magnitude faster and more scalable than the standard optimization techniques like Quadratic Programming(QP) and Second Order Cone Programming (SOCP).

Pros - 1. The paper proposes first of its kind technique to address and use the structured correlation between the output variables in multi-task regression problems 2. The Proximal gradient method proposed on the paper can be used to optimize any kind of convex optimization problems that involve a smooth convex loss and fusion penalty defined on any arbitrary graph structures. 3. The proposed method is faster and more scalable than the standard optimization techniques

Cons - 1. The proximal technique has been compared with the QP and SOCP techniques but not with the subgradient technique as per the illustrations
2. Our results indicate that for small sample sizes, subgradient method can be faster than the Proximal gradient technique.

Not understood - 1. Formulation/proof of the Fusion penalty - We understand hat fusion penalty makes the correlated outputs to share a common set of inputs and hence idenfies a set of inputs that are directly relevant to the outputs instead of having to consider all inputs together. Basically, it helps to exploit the sparsity in the outputs.
2. We did not understand the proofs of lemmas as we're not much familiar with the Optimization theory.

Not implemented - The evaluations for varying values of ρ and signal-to-noise ratio b are not done The implementation of SOCP and QP methods as the concepts are not well understood

6. References

[1] Guillaume Obozinski, Ben Taskar, and Michael I. Jordan. High-dimensional union support recovery in multivariate regression. In NIPS. MIT Press, 2009.

[2] Liang Sun, Jun Liu, Jianhui Chen, and Jieping Ye. Efficient recovery of jointly sparse vectors. In NIPS, 09.

[3] J. Zhu, B. Zhang, E.N. Smith, B. Drees, R.B. Brem, L. Kruglyak, R.E. Bumgarner, and E.E. Schadt. Integrating large-scale functional genomic data to dissect the complexity of yeast regulatory networks. Nature Genetics, 40:854–61, 2008

[4] Han Liu, Mark Palatucci, and Jian Zhang. Blockwise coordinate descent procedures for the multi-task lasso, with applications to neural semantic basis discovery. In International Conference on Machine Learning, 2009

[5] Joumana Ghosn and Yoshua Bengio. Multi-task learning for stock selection. In Advances in Neural Information Processing Systems (NIPS), 1997

[6] Robert Tibshirani and Michael Saunders. Sparsity and smoothness via the fused lasso. J.R.Statist.Soc.B, 67(1):91–108, 2005

[7] Xi Chen, Seyoung Kim, Qihang Lin, Jaime G. Carbonell, Eric P. Xing. Graph -Structured Multi-task Regression and Efficient Optimization Method for General Fused Lasso 2010

7. Appendix

Theorem 1. for any $\mu > 0, f_\mu(B)$ is a convex and continuously differentiable function in B with the gradient:

$$\nabla f_\mu(B) = \Gamma^*(A^*) = A^* C^T \tag{8}$$

To compute $\nabla f_\mu(B)$ and L_μ in the above the theorem we need to know A^* and $||\Gamma||$. We present the closed-form expressions of A^* and $||\Gamma||$ in the following two lemmas.

A1 - Proof of Theorem 1

The $f_\mu(B)$ is a convex function since it is the maximum of a set of functions linear in B.

For the smoothness property, let the function d^* be the Fenchel conjugate of the distance function d which is defined as:

$$d^*(\Theta) = \max_{A \in Q} \langle A, \Theta \rangle - d(A).$$

We want to prove d^* is differential everywhere by showing that the subdifferential ∂d^* of d^* is a singleton set for any Θ

By definition in (equation above), we have, for any Θ and any $A \in Q$:

$$d^*(\Theta) + d(A) \geq \langle A, \Theta \rangle,$$

and the inequality holds as an equality if and only if $A = \underset{A' \in Q}{argmax} \langle A', \Theta \rangle - d\langle A' \rangle$.

By the fact that for a convex and smooth function, the conjugate of the conjugate of a function is the function of itself, we have $d^{**} \equiv d$. Then, (equation above) can be written as:

$$d^*(\Theta) + d^{**}(A) \geq \langle A, \Theta \rangle,$$

and the inequality hold as an equality if and only if $\Theta = \underset{\Theta' \in \mathbb{R}^J}{argmax} \langle A, \Theta' \rangle - d^*(\Theta')$.

Since (the first and second equation) are equivalent, we know that $A = \underset{A' \in Q}{argmax} A'^T \Theta - d(A')$ if and only if $\Theta = \underset{\Theta' \in \mathbb{R}^J}{argmax} \langle A, \Theta' \rangle - d^*(\Theta')$. The latter equality implies that for any Θ':

$$d^*(\Theta') \geq d^*(\Theta) + \langle A, \Theta' - \Theta \rangle,$$

which further means that A is a subgradient of d' at Θ by the definition of subgradient.

Summarizing the above arguments, we conclude that A is a subgradient of d^* at Θ^* if and only if

$$A = \underset{A' \in Q}{argmax} \langle A', \Theta \rangle - d^*(A').$$

Since d is a strongly convex function, this maximization problem in (above equation) has a unique optimal solution, which means the subdifferential $\partial d^* \text{of} d^*$ at any point Θ is a singleton set that contains only A. Therefore, d^* is the differential everywhere and A is its gradient:

$$\Delta d^*(\Theta) = A = \underset{A' \in Q}{argmax} \langle A', \Theta \rangle - d(A').$$

No we return to our original problem of $f_\mu(B)$ and rewrite it as:

$$f_\mu(B) = \max_{A \in Q} \langle A, \Gamma(B) \rangle - \mu d(A) = \mu \max_{A \in Q} [\langle A, \frac{\Gamma(B)}{\mu} \rangle - d(A)] = \mu d^*(\frac{\Gamma(B)}{\mu}$$

Finally, utilizing the equation above and chain rule, we know that $f_\mu(B)$ is continuously differentiable and its gradient take the following form:

$$\Delta f_\mu(B) = \mu \Gamma^*(\Delta d^*(\frac{\Gamma(B)}{\mu})) = \mu \Gamma^*(\underset{A' \in Q}{argmax} [\langle A, \frac{\Gamma(B)}{\mu} \rangle - d(A)] = \Gamma^*(\underset{A' \in Q}{argmax} [\langle A, \Gamma(B) \rangle - \mu d(A)']) = \Gamma^*(A^*). \tag{9}$$

A2 - Proof of Lemma 1 and 2

Lemma 1. Let A^* be the optimal solution of (eq 6 in the text)

$$A^* = S(\frac{BC}{\mu}) \tag{10}$$

Where S is the shrinkage operator defined as follows. *For* $x \in \mathbb{R}, S(x) = xif -1 < x < 1, S(x) = 1ifx \geq 1,$ *and* $S(x) = -1ifx \leq -1$ *For matrix* $A, S(A)$ *is defined as applying* S *on each and every entry of* A. *Proof* by taking the derivative of (eq 6 in teh paper) and setting it to zeros, we obtain $A = \frac{BC}{}$. Then, we project this solution onto the Q and get the optimal solution for A^*.

Lemma 2. $||\Gamma||$ is upper bounded by $||\Gamma||_U \equiv \sqrt{\lambda^2 + 2\gamma^2 \mathbf{max}_k d_k}$ where

$$d_k = \sum_{e \in E \text{ s.t. } e \text{ incident on } k} (\tau(r_e))^2 \tag{11}$$

for $k \in V$ on graph G; and this bound is tight. According to the definition of $||\Gamma||$, we have:

$$||\Gamma|| \equiv \max_{||B||_F=1} ||\Gamma(B)_F = \max_{||B||_F=1} ||\lambda B, \gamma BH||_F = \max_{||B||_F=1} \sqrt{\lambda^2 ||B||_F^2 + \gamma^2 ||BH||_F^2} = \max_{||B||_F=1} \sqrt{\lambda^2 + \gamma^2 ||BH||_F^2} \tag{12}$$

Therefore, to bound $||\Gamma||$, we only need to find an upper bound for the $\max_{||B||_F=1} ||BH||_F^2$. According to the formulation of the matrix H, we have

$$||BH||_F^2 = \sum_{e=(m,l) \in E} (\tau(r_{ml}))^2 \sum_j (\beta_{jm} - sign(r_{ml})\beta_{jl})^2 \tag{13}$$

It is well known that $(a-b)^2 \leq 2a^2 + 2b^2$ and the inequality holds as equality if and only if $a = -b$. Using this simple inequality, for each edge $e = (m, l) \in E$, the summation $\sum_j (\beta_{jm} - sign(r_{ml}\beta_{jl})^2$ is upper-bounded by $\sum_j (2\beta_{jm}^2 + 2\beta_{jl}^2) = 2||\beta_m||_2^2 + 2||\beta_l||_2^2$. Here, the vectors β_m and β_l are the m-th and l-th columns of B. The right-hand side of (equation above) can be further bounded as:

$$||BH||_F^2 \leq \sum_{e=(m,l) \in E} 2(\tau(r_{ml}))^2 (||\beta_m||_2^2 + ||\beta_l||_2^2) = \sum_{k \in V} (\sum_{eincidentonk} 2(\tau(r_e))^2) ||\beta_k||_2^2 = \sum_{k \in V} 2d_k ||\beta_k||_2^2, \tag{14}$$

where d_k is defined in (equation 9 from paper). Note that the first inequality is tight, and that the first equality can be obtained simply by changing the order of summations.
By definition pf Frobenius norm, $||B||_F^2 = \sum_k ||\beta_k||_2^2$. Hence,

$$\max_{||B||_f=1} ||BH||_F^2 \leq \max_{||B||_F=1} \sum_k 2d_k ||\beta_k||_2^2 = 2 \max_k d_k, \tag{15}$$

where the maximum is achieved by setting the β_k corresponding to the largest d_k to be a unit vector and setting other β_k 's to be zero vectors.
In summary, $||\Gamma||$ can be tightly upper bounded as:

$$||\Gamma|| = \max_{||B||_f=1} ||\Gamma(B)||_F = \max_{||B||_f=1} \sqrt{\lambda^2 + \gamma^2 ||BH||_F^2} = \sqrt{\lambda^2 + \gamma^2 \max_{||B||_F=1} ||BH||_F^2} = \sqrt{\lambda^2 + 2\gamma^2 \max_{||B||_F=1} d_k} \equiv ||\Gamma||_U \tag{16}$$

A3 - The ℓ_1 and ℓ_1/ℓ_2 Regularized Multi-task Regression problem can be formulated as follows -

This Lasso in technique in Eq. (2) does not offer any mechanism for a joint estimation of the parameters for the multiple outputs. To overcome this challenge, a mixed norm (e.g. l1/l2) regularization is used for multiple tasks when the tasks share a common set of relevant covariates. l1/l2 regularization encourages the relevant covariates to be shared across output variables and finds estimates in which only few covariates have non-zero regression coefficients for one or more of the K output variables. The corresponding optimization problem is defined as follows

$$\hat{B}^{l_1/l_2} = \text{argmin}\frac{1}{2}||Y - XB||_F^2 + \lambda||B||_{1,2} \tag{17}$$

where $||B||$ = Although it jointly estimates parameters for multiple output variables, it assumes that ALL tasks are equally related the inputs. But as described above, in real world problems, some tasks are more closely related (share common covariates more likely) than other tasks. Thus, there is a complex correlation structure in the outputs which cannot be incorporated using l1/l2 regularization.

A4 - Pearson's correlation In statistics, the Pearson correlation also referred to as Pearson's r, is a statistic that measures linear correlation between two variables X and Y. Given a pair of random variables (X, Y), Pearson's correlation is defined as

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{18}$$

It has a value between +1 and 1 A value of +1 is total positive linear correlation, 0 is no linear correlation, and 1 is total negative linear correlation.

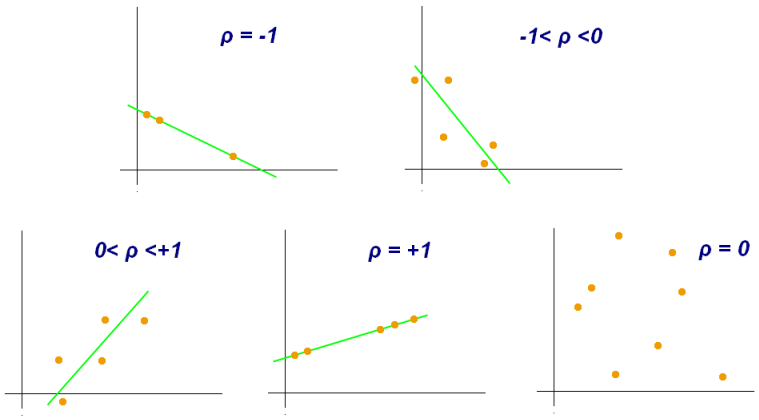

Figure 5: Examples of scatter diagrams with different values of correlation coefficient (p)

A5 - Fusion Penalty – is a novel penalty function which "encourages" the highly correlated outputs to share a common set of "relevant inputs". Explicitly stated, fusion penalty "fuses" the regression coefficients **across** correlated outputs using the "weighted connectivity" of the output graph as a guide. Overall effect of penalty function is that "it allows us to identify a small set of input factors relevant to the dense subgraphs of outputs a shown in the figure below:

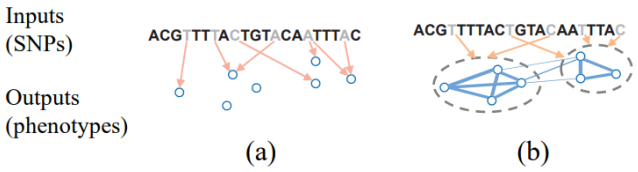

Figure 6: Illustration of multi-task regression with a) Lasso and b)Graph guided fused lasso

Fusion penalty has been widely used for sparse learning problems like Fused Lasso, network learning, etc.

A6 - Code

```
# -*- coding: utf-8 -*-
"""Graph Guided Lasso.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/11L7jQqcJDto6_fEH7v0fjx16NPOoJk2J

##An Implementation of General Fused Lasso
Based on - Graph -Structured Multi-task Regression and Efficient Optimization Method for General Fused Lasso;
Authors    - Xi Chen, Seyoung Kim, Qihang Lin, Jaime G. Carbonell, Eric P. Xing\\
Source     - [v1] Thu, 20 May 2010 on arXiv:1005.3579 https://arxiv.org/abs/1005.3579
"""

# Commented out IPython magic to ensure Python compatibility.
import matplotlib.pyplot as plt
import numpy as np
import time
import pylab
from sklearn.linear_model import MultiTaskLasso, Lasso
from sklearn.model_selection import train_test_split
rng = np.random.RandomState(42)
# %matplotlib inline

"""##GFLasso with Proximal Gradiet function
Model design for general fused lasso along with proximal gradient method
"""

class GraphFusedLasso():
    """
    Graph-Guided Fused Lasso
    """
    def __init__(
        self, G, theta_val=0, X_b=0, lr=0, n_iter=0,max_iterations=50, lambda_val=1.0, gamma_val=1.0, epsilon=1.0, tol=10**-6,
        no_bias=False, verbose=False, grad_func=1,
    ):
        self.G = G
        self.max_iterations = max_iterations
        self.lambda_val = lambda_val
        self.gamma_val = gamma_val
        self.epsilon = epsilon
        self.tol = tol
        self.no_bias = no_bias
        self.verbose = verbose
```

```
        self.grad_func='proximal'
        self.theta_val = theta_val


    def fit(self, X, y):
        #adding bias for y_pred
#        print(grad_func)
        if grad_func == 1:
          if not self.no_bias:
              X = self.bias_padding(X)
        N, J = X.shape
        tasks = y.shape[1] #K
        edges = np.nonzero(self.G)[0].shape[0] #E

        # Initialize vertex-edge incident matrix H with KxE
        H = np.zeros((tasks, edges))
        #Fill H as per Sec 4.1 definition
        for k in range(tasks):
            for e, (m, l) in enumerate(zip(*np.nonzero(self.G ))):
                if k == m:
                    H[k, e] = np.abs(self.G[m, l])
                elif k == l:
                    H[k, e] = -np.sign(self.G[m, l]) * np.abs(self.G[m, l])

        I = np.identity(tasks)
        # lasso and graph-guided fusion penalty functions as ||BC||_1, with C = ( I , H )
        C = np.concatenate((self.lambda_val * I, self.gamma_val * H), axis=1)
        #d(A)   1  /2*||A||^2_F; D = 1/2*J(K+E) Sec 4.2
        D = 0.5 * J * (tasks + edges)
        #||  ||_U, mapping from JxK to Jx(K+|E|)
        d_k = np.sum(self.G**2, axis=1)
        #mu defined to achieve best convergance rate
        mu = self.epsilon / (2. * D)
        #eigenvector X^TX
        eigen_value, _ = np.linalg.eig(np.dot(X.T, X))
        #Upper bounded Lipschitz constant Eq 12 L_U =  max (XTX) + (  ^2 + 2  ^2 max V  d_k)/
        L_u = np.max(eigen_value) + (self.lambda_val**2 + 2 * self.gamma_val**2 * np.max(d_k)) / mu

        # Proximal Gradient Function, Algorithm 1
        W_prev = np.zeros((J, tasks))
        B = np.ones((J, tasks))
        Z_prev = 0.
        B_prev = 0.
        coefdiff_history = []
        loss_history = []
        if grad_func == 1:
          B, coefdiff_history, loss_history = self.proximal_gradient(W_prev, B_prev, B, X, y, C, mu, L_u, Z_prev, self.max_iterations, loss_history, coefdiff_history)
        #Stochastic Gradient Method added only for the Computation Comparison purpose.
        #This method cannot be used for generating useful output
        elif grad_func == 2:
          self.theta,self.cost_history = stochastic_grad_desc(X_b,y,theta,lr,n_iter)
        #Stochastic Gradient Method added only for the Computation Comparison purpose.
        #This method cannot be used for generating useful output
        elif grad_func == 3:
          w_init = np.random.randn(X.shape[1])
          w,b = stochastic_subgrad(X,y,w_init,b = 10,tradeoff=1, itr=n_iter)
        else:
          print("Define Gradient Function to use")
```

```
103
104
105         self.coef_ = B.T[:, :-1] if not self.no_bias else B.T
106         self.intercept_ = B.T[:, -1] if not self.no_bias else np.zeros(tasks)
107         self.coefdiff_history_ = coefdiff_history
108         self.loss_history_ = loss_history
109         return self
110
111     def gflasso_prediction(self, X):
112         #model definition Eq 1
113         y_pred = np.dot(X, self.coef_.T) + self.intercept_
114         return y_pred
115
116     def bias_padding(self, X):
117         b = np.ones((X.shape[0], 1), dtype=X.dtype)
118         return np.concatenate((X, b), axis=1)
119
120     #Eq 14 as graph fusion penalty
121     def graph_fusion_penalty(self, X, y, B):
122         #vector l2-norm
123         loss = 0.5 * np.sum((y - np.dot(X, B))**2)
124         loss += self.lambda_val * np.linalg.norm(B, ord=1)
125         for e, (m, l) in enumerate(zip(*np.nonzero(self.G))):
126             r = self.G[m, l]
127             s = np.sign(r)
128             loss += self.gamma_val * np.abs(r) * np.sum(np.abs(B[:, m] - s * B[:, l])
    )
129         return loss
130
131     def proximal_gradient(self, W_prev, B_prev, B, X, y, C, mu, L_u, Z_prev, itr,
    loss_history, coefdiff_history):
132         for itr in range(self.max_iterations):
133         #derivaative of Optimal solution A*
134         #lemma 1
135             S_A = shrinkage_operator(np.dot(W_prev, C) / mu)
136             #Eq 11  f  (B) = X^T(XB     Y) +  A C ^TT
137             grad_f_b = np.dot(X.T, (np.dot(X, W_prev) - y)) + np.dot(S_A, C.T)
138
139             # Gradient descent step
140             B = threshold_offset(W_prev - grad_f_b / L_u, self.lambda_val / L_u)
141
142             # Step 3
143             Z = Z_prev - 0.5 * (itr + 1) * grad_f_b / L_u
144
145             # Step 4
146             W_prev = ((itr + 1) / (itr + 3)) * B + (2 / (itr + 3)) * Z
147
148             # Check termination condition
149             loss = self.graph_fusion_penalty(X, y, B)
150             # fusion penalty defined on inputs ordered in time as a chain
151             #i.e., special case of graph_fusion_penalty
152             special_case_penalty = np.mean(np.abs(B - B_prev)) if itr > 0 else np.inf
153             loss_history.append(loss)
154             coefdiff_history.append(special_case_penalty)
155
156             if self.verbose:
157                 print("Iter {0}: Fusion Penalty = {1:e}, Loss = {2:e}".format(
158                     itr + 1, special_case_penalty, loss))
159             if special_case_penalty <= self.tol:
160                 #print(itr)
161                 break
162             B_prev = B
163             Z_prev = Z
164
```

```
        return  B, coefdiff_history, loss_history

def shrinkage_operator(array):
    #For x     R, S(x) = x if   1   < x < 1,
    #S(x) = 1 if x     1, and S(x) =   1   if x       1
    array = np.where(array >= 1., 1., array)  # if x >= 1
    array = np.where(array <= -1., -1., array)  # if x <= -1
    return array

def threshold_offset(array, lambda_val):
    array_new = np.zeros_like(array)
    array_new[np.where(array > lambda_val)] = array[np.where(array > lambda_val)] -
    lambda_val
    array_new[np.where(array < -lambda_val)] = array[np.where(array < -lambda_val)] +
     lambda_val
    return array_new

"""##Stochastic Gradient Descent
For comparison purpose only
"""

def fn_cost(theta,X,y):
        predictions = X.dot(theta)
        cost = (1/2*len(y)) * np.sum(np.square(predictions-y))
        return cost

def stochastic_grad_desc(X,y,theta,learning_rate=0.01,itr=10):
        cost_history = np.zeros(itr)
        for it in range(itr):
            cost =0.0
            for i in range(len(y)):
                rand_idx = np.random.randint(0,len(y))
                X_i = X[rand_idx,:].reshape(1,X.shape[1])
                y_i = y[rand_idx,:].reshape(1,y.shape[1])
                pred = np.dot(X_i,theta)

                theta = theta -(1/len(y))*learning_rate*( X_i.T.dot((pred - y_i)))
                cost += fn_cost(theta,X_i,y_i)
            cost_history[it]  = cost

        return theta, cost_history

"""##Stochastic Sub-Gradient
For comparison purpose only
"""

def add_subgrad(w, subgradient_w):
    """
    total_w = w + C *      (-y*x)

    """
    return w + subgradient_w

def calc_subgradient(x, y, w, b, tradeoff):

    subgrad_w = 0
    subgrad_b = 0

    # sum over all subgradients of hinge loss for a given samples x,y
    for xi, yi in zip(x,y):
        fxi = np.dot(w.T, xi) + b

        threshold_val = yi * fxi

```

```
        if threshold_val < 1:
            subgrad_w += - yi*xi
            subgrad_b += -1 * yi
        else:
            subgrad_w += 0
            subgrad_b += 0

    # multiply by C after summation of all subgradients for a given samples of x,y
    subgrad_w = tradeoff * subgrad_w
    subgrad_b = tradeoff * subgrad_b
    return (add_subgrad(w, subgrad_w), subgrad_b)

def stochastic_subgrad(X, y, w, b, tradeoff, itr=50):

    for t in range(1, itr+1):
        learning_rate = 1/t
        for i in range(0, y.shape[1]):
          #print(y[:,i])
          sub_grads = calc_subgradient(X,y[:,i], w, b, tradeoff)

        # update weights
          w = w - learning_rate * sub_grads[0]

        # update bias
          b = b - learning_rate * sub_grads[1]
    return (w,b)

"""##Initialization
Initializing Graph G, X, y, number of samples N, number of features J and number of tasks K
"""

N, J, K = 1000, 30, 40 # N = number of samples, J = features, K = tasks
n_relevant_features = 5
coef = np.zeros((K, J))
times = np.linspace(0, 2 * np.pi, K // 2)
for k in range(n_relevant_features):
    coef[:K // 2, k] = np.sin((1. + rng.randn(1)) * times + 3 * rng.randn(1))
    coef[K // 2:, k] = coef[:K // 2, k]

X = rng.randn(N, J)
#X=np.random.normal(0, 1, size=(N, J))
y = np.dot(X, coef.T) + rng.randn(N, K)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7,
                                                    random_state=42)

G = np.identity(K)

for j in range(K // 2):
    G[j, K // 2 + j] = G[K // 2 + j, j] = 1.0

"""##GFLasso Model with Proximal Gradient Method
Also computes false positive rate and true positive rate for each relevant feature
"""

grad_func=1
gflasso_model = GraphFusedLasso(G, verbose=True, max_iterations=1000, lambda_val=2., epsilon=1.).fit(X_train, y_train)
gflasso_coefficients_ = gflasso_model.coef_

rho = 0.5
gflasso_coefficients = np.zeros_like(gflasso_coefficients_)
gflasso_coefficients[gflasso_coefficients_ < rho] = 0

```

```
from sklearn.metrics import roc_auc_score, roc_curve, auc

y_pred=gflasso_model.gflasso_prediction(X_test)
y_test_classes = np.zeros_like(y_pred)
cutoff = 0.1
y_test_classes[y_test > cutoff] = 1
fpr_gfl = dict()
tpr_gfl = dict()

for i in range(n_relevant_features):
    fpr_gfl[i], tpr_gfl[i], _ = roc_curve(y_test_classes[:, i], y_pred[:,i])
    auc = roc_auc_score(y_test_classes[:, i], y_pred[:,i] )
    print("Area under ROC curve for relevant feature at index {0}: {1:e}".format(i,
    auc))

"""##Lasso Model
Also computes false positive rate and true positive rate for each relevant feature
"""

lasso_model = Lasso(alpha=0.5).fit(X, y)
lasso_coefficients = lasso_model.coef_

fpr_lasso = dict()
tpr_lasso = dict()

y_pred_lasso = lasso_model.predict(X_test)[:,1]

for i in range(n_relevant_features):
    fpr_lasso[i], tpr_lasso[i], _ = roc_curve(y_test_classes[:, i], y_pred_lasso[:])

"""##MultiTaskLasso Model
Also computes false positive rate and true positive rate for each relevant feature
"""

multi_task_model = MultiTaskLasso(alpha=1.).fit(X, y)
multi_task_lasso_coefficients = multi_task_model.coef_
fpr_l1l2 = dict()
tpr_l1l2 = dict()

y_pred_l1l2 = multi_task_model.predict(X_test)[:,1]

for i in range(n_relevant_features):
    fpr_l1l2[i], tpr_l1l2[i], _ = roc_curve(y_test_classes[:, i], y_pred_l1l2[:])

"""##ROC Curve
ROC Curve for GFLasso, Lasso and MultiTaskLasso Models
"""

from matplotlib import pyplot as plt
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_lasso[2], tpr_lasso[2], label='Lasso')
plt.plot(fpr_l1l2[2], tpr_l1l2[2], label='l1l2')
plt.plot(fpr_gfl[2], tpr_gfl[2], label='GFLasso')

plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
from google.colab import files
plt.savefig("ROC_idx_2.png")
files.download("ROC_idx_2.png")
plt.show()

"""##Regression Coefficients for all models"""
```

```
352
353 fig = plt.figure(figsize=(16, 5))
354 plt.subplot(1, 4, 1)
355 plt.imshow(coef)
356 plt.xlabel('Feature')
357 plt.ylabel('Task)')
358 plt.text(10, 5, 'Ground truth')
359 plt.subplot(1, 4, 2)
360 plt.imshow(lasso_coefficients)
361 plt.xlabel('Feature')
362 plt.text(10, 5, 'Lasso')
363 plt.subplot(1, 4, 3)
364 plt.imshow(multi_task_lasso_coefficients)
365 plt.xlabel('Feature')
366 plt.text(10, 5, 'MultiTaskLasso')
367 plt.subplot(1, 4, 4)
368 plt.imshow(gflasso_coefficients_)
369 plt.xlabel('Feature')
370 plt.text(10, 5, 'GraphFusedLasso')
371 from google.colab import files
372 plt.savefig("Coeffs.png")
373 files.download("Coeffs.png")
374 plt.show()
375
376 fig = plt.figure(figsize=(14, 5))
377 lw = 2
378 feature_to_plot = 4
379 plt.plot(coef[:, feature_to_plot], linewidth=lw,
380          label='Ground truth')
381 plt.plot(lasso_coefficients[:, feature_to_plot], linewidth=lw,
382          label='Lasso')
383 plt.plot(multi_task_lasso_coefficients[:, feature_to_plot], linewidth=lw,
384          label='MultiTaskLasso')
385 plt.plot(gflasso_coefficients_[:, feature_to_plot], linewidth=lw,
386          label='GraphFusedLasso')
387 plt.legend(loc='lower center')
388 plt.axis('tight')
389 plt.title("Feature index: {}".format(feature_to_plot))
390 plt.ylim([-1.1, 1.1])
391
392 from google.colab import files
393 plt.savefig("Feature.png")
394 files.download("Feature.png")
395 plt.show()
396
397 """##Heatmap of relevant feature"""
398
399 import seaborn as sns
400
401 import pandas as pd
402 X=pd.DataFrame(X)
403 # taking all rows but only 5 columns
404 df_small = X.iloc[:,:n_relevant_features]
405
406 correlation_mat = df_small.corr()
407
408 sns.heatmap(correlation_mat, annot = True)
409 from google.colab import files
410 plt.savefig("Heatmap.png")
411 files.download("Heatmap.png")
412 plt.show()
413
414 """##Evaluation in terms of J
415 varying value of J from 1000 to 10000 with step size of 1000
```

```
"""

import time

comp_time_J_prox = []
comp_time_J_sgd = []
comp_time_J_subg = []
for J in range(1000,10000,1000):

  N, K = 1000, 40 # N = number of samples, J = features, K = tasks
  n_relevant_features = 5
  coef = np.zeros((K, J))
  c = np.linspace(0, 2 * np.pi, K // 2)
  for k in range(n_relevant_features):
      coef[:K // 2, k] = np.sin((1. + rng.randn(1)) * c + 2 * rng.randn(1))
      coef[K // 2:, k] = coef[:K // 2, k]

  #X = rng.randn(N, J)
  X=np.random.normal(0, 1, size=(N, J))
  y = np.dot(X, coef.T) + rng.randn(N, K)
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7,
                                                      random_state=42)

  G = np.identity(K)
  grad_func=1
  for j in range(K // 2):
      G[j, K // 2 + j] = G[K // 2 + j, j] = 1.0


  print("J = {0} for Proximal Gradient Function".format(J))
  start_time = time.time()
  gflasso_model = GraphFusedLasso(G, verbose=True, max_iterations=1000, lambda_val
    =2., epsilon=1.,grad_func=1).fit(X, y)
  end_time = time.time()
  comp_time_J_prox.append(end_time - start_time)
  grad_func=2

  print("J = {0} for Stochastic Gradient Function".format(J))
  start_time = time.time()
  lr =0.5
  n_iter = 50
  theta = np.random.randn(J+1,1)
  X_b = np.c_[np.ones((len(X),1)),X]
  gflasso_model = GraphFusedLasso(G, theta,X_b,lr,n_iter=n_iter, verbose=True,
    max_iterations=1000, lambda_val=2., epsilon=1,grad_func=2).fit(X, y)
  end_time = time.time()
  comp_time_J_sgd.append(end_time - start_time)

  print("J = {0} for Stochastic Sub-Gradient Function".format(J))
  start_time = time.time()
  gflasso_model = GraphFusedLasso(G, verbose=True,grad_func=3, n_iter=50).fit(X, y)
  end_time = time.time()
  comp_time_J_subg.append(end_time - start_time)
print(comp_time_J_prox)
print(comp_time_J_subg)

# Random X
comp_time_J_prox=[24.543862342834473, 151.35878896713257, 232.7870545387268,
    325.7700753211975, 437.12480187416077, 570.6482968330383, 737.5211029052734,
    926.0571410655975, 1143.4936137199402]
comp_time_J_subg=[18.299408435821533, 36.21959400177002, 64.98038983345032,
    107.28624105453491, 163.075453042984, 244.21183276176453, 343.64225935935974,
    475.861576795578, 653.4304647445679]

```

```
#Normal Distribution X
#comp_time_J_prox = [27.635534524917603, 156.98772144317627, 237.92661786079407,
    338.9670236110687, 461.8965120315552, 603.6267650127411, 767.9135549068451,
    1009.3803911209106, 1247.4318089485168]
#comp_time_J_subg = [20.421093702316284, 38.149115324020386, 68.76089596748352,
    112.44459652900696, 175.34537506103516, 265.1817409992218, 387.21791982650757,
    535.1038284301758, 716.0683045387268]

a = [pow(10, i) for i in range(10)]
fig = plt.figure()

J_new=[]
for i in range(1000,10000,1000):
  J_new.append(i)

plt.plot(J_new, comp_time_J_prox, label='Prox_grad', linestyle='--', marker='o')
plt.plot(J_new, comp_time_J_subg, label='SubGrad', linestyle='--', marker='o')
plt.yscale("log")
plt.legend(loc='best')
from google.colab import files
plt.savefig("Evaluation_Varying_J_Random_X.png")
files.download("Evaluation_Varying_J_Random_X.png")
plt.show()

"""##Evaluation in terms of N
varying value of N from 500 to 10000 with step size of 1000
"""

comp_time_N_prox = []
comp_time_N_sgd = []
comp_time_N_subg = []
for N in range(500,10000,500):

  J, K = 30, 40 # N = number of samples, J = features, K = tasks
  n_relevant_features = 5
  coef = np.zeros((K, J))
  c = np.linspace(0, 2 * np.pi, K // 2)
  for k in range(n_relevant_features):
      coef[:K // 2, k] = np.sin((1. + rng.randn(1)) * c + 2 * rng.randn(1))
      coef[K // 2:, k] = coef[:K // 2, k]

  #X = rng.randn(N, J)
  X=np.random.normal(0, 1, size=(N, J))
  y = np.dot(X, coef.T) + rng.randn(N, K)
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7,
                                                      random_state=42)

  G = np.identity(K)
  grad_func=1
  for j in range(K // 2):
      G[j, K // 2 + j] = G[K // 2 + j, j] = 1.0
  start_time = time.time()
  print("N = {0} for Proximal Gradient Function".format(N))
  gflasso_model = GraphFusedLasso(G, verbose=True, max_iterations=1000, lambda_val
    =2., epsilon=1.,grad_func=1).fit(X, y)
  end_time = time.time()
  comp_time_N_prox.append(end_time - start_time)
  grad_func=2

  print("N = {0} for Stochastic Gradient Function".format(N))
  start_time = time.time()
  lr =0.5
  n_iter = 50
  theta = np.random.randn(J+1,1)
```

```
  X_b = np.c_[np.ones((len(X),1)),X]
  gflasso_model = GraphFusedLasso(G, theta,X_b,lr,n_iter=n_iter, verbose=True,
    max_iterations=1000, lambda_val=2., epsilon=1,grad_func=2).fit(X, y)
  end_time = time.time()
  comp_time_N_sgd.append(end_time - start_time)

  print("N = {0} for Stochastic Sub-Gradient Function".format(N))
  start_time = time.time()
  gflasso_model = GraphFusedLasso(G, verbose=True,grad_func=3, n_iter=50).fit(X, y)
  end_time = time.time()
  comp_time_N_subg.append(end_time - start_time)
print(comp_time_N_prox)
print(comp_time_N_subg)

# Random X
#comp_time_N_prox = [1.6524100303649902, 1.0764613151550293, 0.8702692985534668,
    0.7575669288635254, 0.7244038581848145, 0.708141565322876, 0.6475667953491211,
    0.6390447616577148, 0.6226475238800049, 0.6034164428710938, 0.6401464939117432,
    0.6154055595397949, 0.6249487400054932, 0.5878493785858154, 0.5842859745025635,
    0.6210455894470215, 0.6097066402435303, 0.5902907848358154, 0.5882308483123779]
#comp_time_N_subg = [0.6896660327911377, 1.450862169265747, 2.0305209159851074,
    2.7442500591278076, 3.486647129058838, 4.114997863769531, 4.959404945373535,
    5.4457550048828125, 6.383319139480591, 6.859850168228149, 7.523995399475098,
    8.253390073776245, 9.128736019134521, 9.697450160980225, 10.434390544891357,
    11.100789308547974, 11.551125288009644, 12.4006028175354, 13.446029901504517]

#Normal Distribution X
comp_time_N_prox = [1.7502226829528809, 1.0987131595611572, 0.8918313980102539,
    0.8624117374420166, 0.8024282455444336, 0.6992778778076172, 0.6947231292724609,
    0.7428948879241943, 0.6163134574890137, 0.6007742881774902, 0.6111116409301758,
    0.602060079574585, 0.5894467830657959, 0.5747578144073486, 0.5769977569580078,
    0.5957958698272705, 0.6129364967346191, 0.6109001636505127, 0.6175305843353271]
comp_time_N_subg = [0.706697940826416, 1.4935050010681152, 2.1821913719177246,
    2.6953771114349365, 3.381666421890259, 4.15513801574707, 5.108561277389526,
    5.522367238998413, 6.096566200256348, 6.858453989028931, 7.681184530258179,
    8.403875350952148, 8.831019639968872, 9.525176525115967, 10.188826560974121,
    11.004908084869385, 11.664239406585693, 12.511788606643677, 13.351653099060059]

a = [pow(10, i) for i in range(10)]
fig = plt.figure()

N_new=[]
for i in range(500,10000,500):
  N_new.append(i)

plt.plot(N_new, comp_time_N_prox, label='Prox_grad', linestyle='--', marker='o')
plt.plot(N_new, comp_time_N_subg, label='SubGrad', linestyle='--', marker='o')

plt.yscale("log")
plt.legend(loc='best')
from google.colab import files
plt.savefig("Evaluation_Varying_N_Normal_X.png")
files.download("Evaluation_Varying_N_Normal_X.png")
plt.show()

"""##Evaluation in terms of K
varying value of K from 1000 to 8000 with step size of 1000
"""

comp_time_K_prox = []
comp_time_K_sgd = []
comp_time_K_subg = []
for K in range(1000,8000,1000):

```

```
  N, J = 1000, 30 # N = number of samples, J = features, K = tasks
  n_relevant_features = 5
  coef = np.zeros((K, J))
  c = np.linspace(0, 2 * np.pi, K // 2)
  for k in range(n_relevant_features):
      coef[:K // 2, k] = np.sin((1. + rng.randn(1)) * c + 2 * rng.randn(1))
      coef[K // 2:, k] = coef[:K // 2, k]

  X = rng.randn(N, J)
  #X=np.random.normal(0, 1, size=(N, J))
  y = np.dot(X, coef.T) + rng.randn(N, K)
  X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7,
                                                      random_state=42)

  G = np.identity(K)
  grad_func=1
  for j in range(K // 2):
      G[j, K // 2 + j] = G[K // 2 + j, j] = 1.0
  start_time = time.time()
  print("K = {0} for Proximal Gradient Function".format(K))
  gflasso_model = GraphFusedLasso(G, verbose=True, max_iterations=1000, lambda_val
    =2., epsilon=1.,grad_func=1).fit(X, y)
  end_time = time.time()
  comp_time_K_prox.append(end_time - start_time)
  grad_func=2

  print("K = {0} for Stochastic Gradient Function".format(K))
  start_time = time.time()
  lr =0.5
  n_iter = 50
  theta = np.random.randn(J+1,1)
  X_b = np.c_[np.ones((len(X),1)),X]
  gflasso_model = GraphFusedLasso(G, theta,X_b,lr,n_iter=n_iter, verbose=True,
    max_iterations=1000, lambda_val=2., epsilon=1,grad_func=2).fit(X, y)
  end_time = time.time()
  comp_time_K_sgd.append(end_time - start_time)

  print("K = {0} for Stochastic Sub-Gradient Function".format(K))
  start_time = time.time()
  gflasso_model = GraphFusedLasso(G, verbose=True,grad_func=3, n_iter=50).fit(X, y)
  end_time = time.time()
  comp_time_K_subg.append(end_time - start_time)
print(comp_time_K_prox)
print(comp_time_K_subg)

# Random X
#comp_time_K_prox = [121.5137665271759, 352.7910442352295, 724.7827446460724,
    1318.319215297699, 2183.0467767715454, 3340.735093355179, 4900.298081159592]
#comp_time_K_subg = [17.646416425704956, 95.27220606803894, 275.81103587150574,
    608.4826486110687, 1161.9301245212555, 1950.5011146068573, 3049.594600915909]

#Normal Distribution X
comp_time_K_prox = [113.9111864566803, 337.96793007850647, 690.1005334854126,
    1258.1055722236633, 2082.5863206386566, 3180.7012462615967, 4724.101638555527]
comp_time_K_subg = [16.972026824951172, 93.48991560935974, 264.95730996131897,
    585.7650554180145, 1104.0863575935364, 1890.8375437259674, 2961.358941078186]

a = [pow(10, i) for i in range(10)]
fig = plt.figure()

K_new=[]
for i in range(1000,8000,1000):
  K_new.append(i)
```

```
plt.plot(K_new, comp_time_K_prox, label='Prox_grad', linestyle='--', marker='o')
plt.plot(K_new, comp_time_K_subg, label='SubGrad', linestyle='--', marker='o')
plt.yscale("log")
plt.legend(loc='best')
from google.colab import files
plt.savefig("Evaluation_Varying_K_Normal_X.png")
files.download("Evaluation_Varying_K_Normal_X.png")
plt.show()
```