```
clear all
close all

%% Adding the paths
addpath('C:\Users\Cristina\Documents\GitHub\OrganizedFiles\Optimizers'); %Folder↙
conatining the yalmip tools
addpath('C:↙
\Users\Cristina\Documents\GitHub\OrganizedFiles\DataSets\Comparison_datasets\'); %↙
Folder containing the comparison datasets
addpath('C:\Users\Cristina\Documents\GitHub\OrganizedFiles\DataSets\'); %Folder↙
containing the training and verification dataset
addpath('C:↙
\Users\Cristina\Documents\GitHub\OrganizedFiles\GeneratingKernels\Results'); %Folder↙
conatining the heat kernel coefficients
path = 'C:↙
\Users\Cristina\Documents\GitHub\OrganizedFiles\Graph&DictLearning\G&D_fromGraphCorrec↙
t\Results\'; %Folder containing the results to save

flag = 1;

switch flag
    case 1 %Dorina
        load DataSetDorina.mat
        load ComparisonDorina.mat
    case 2 %Uber
        load DataSetUber.mat
        load ComparisonUber.mat
    case 3 %Cristina
        load DatasetLF.mat
        load ComparisonLF.mat;
    case 4 %1 Heat kernel
        load DataSetHeat.mat;
        load ComparisonHeat.mat;
        load LF_heatKernel.mat;
end

switch flag
    case 1 %Dorina
        Y = TrainSignal;
        K = 20;
        param.S = 4;  % number of subdictionaries
        param.epsilon = 0.05; % we assume that epsilon_1 = epsilon_2 = epsilon
        degree = 20;
        ds = 'Dataset used: Synthetic data from Dorina';
        ds_name = 'Dorina';
        param.percentage = 15;
        param.thresh = param.percentage + 60;
    case 2 %Uber
        Y = TrainSignal;
        K = 15;
        param.S = 2;  % number of subdictionaries
        param.epsilon = 0.2; % we assume that epsilon_1 = epsilon_2 = epsilon
        ds = 'Dataset used: data from Uber';
        ds_name = 'Uber';
        param.percentage = 8;
        param.thresh = param.percentage + 6;
    case 3 %Cristina
        Y = TrainSignal;
        K = 15;
        param.S = 2;  % number of subdictionaries
        param.epsilon = 0.02; % we assume that epsilon_1 = epsilon_2 = epsilon
        degree = 15;
        ds = 'Dataset used: data from Cristina';
        ds_name = 'Cristina';
        param.percentage = 8;
        param.thresh = param.percentage + 6;
    case 4 %Heat kernel
        Y = TrainSignal;
        K = 15;
        param.S = 1;  % number of subdictionaries
        param.epsilon = 0.2; % we assume that epsilon_1 = epsilon_2 = epsilon
        ds = 'Dataset used: data from Heat kernel';
        ds_name = 'Heat';
        param.percentage = 8;
        param.thresh = param.percentage + 6;
end

param.N = size(Y,1); % number of nodes in the graph
param.J = param.N * param.S; % total number of atoms
param.K = K*ones(1,param.S); %[20 20 20 20]; % polynomial degree of each subdictionary
param.c = 1; % spectral control parameters
param.mu = 1;%1e-2; % polynomial regularizer paremeter
param.y = Y; %signals
param.y_size = size(param.y,2);
param.T0 = 6; %sparsity level (# of atoms in each signals representation)
param.max = 0;
alpha = 2; %gradient descent parameter, it decreases with epochs

%% Initialize the kernel coefficients
temp = comp_alpha;
comp_alpha = zeros(K+1,param.S);
for i = 1:param.S
    comp_alpha(:,i) = temp((K+1)*(i-1) + 1:(K+1)*i);
end

if flag ~= 4
    for i = 1:param.S
        param.alpha{i} = comp_alpha(:,i);
    end
else
    param.t(1) = 2; %heat kernel coefficients
    param.t(2) = 1; %this heat kernel will be inverted to cover high frequency↙
components
    param.alpha = generate_coefficients(param);
    disp(param.alpha);
end

%% Obtain the initial Laplacian and eigenValues for comparison
out_L = diag(sum(comp_W,2)) - comp_W; % combinatorial Laplacian
comp_Laplacian = (diag(sum(comp_W,2)))^(-1/2)*out_L*(diag(sum(comp_W,2)))^(-1/2); %↙
```

```
normalized Laplacian
[comp_eigenMat, comp_eigenVal] = eig(comp_Laplacian);
[comp_lambdaSym,comp_indexSym] = sort(diag(comp_eigenVal));

comp_lambdaPowerMx(:,2) = comp_lambdaSym;
for i = 1:K+1
    comp_lambdaPowerMx(:,i) = comp_lambdaPowerMx(:,2).^(i-1);
    comp_Laplacian_powers{i} = comp_Laplacian^(i-1);
end

comp_ker = zeros(param.N,param.S);
for i = 1 : param.S
    for n = 1:param.N
        comp_ker(n,i) = comp_ker(n,i) + comp_lambdaPowerMx(n,:)*comp_alpha(:,i);
    end
end

%% Initialize W:

[param.Laplacian, initial_W] = init_by_weight(param.N);
initial_Laplacian = param.Laplacian;

[param.eigenMat, param.eigenVal] = eig(param.Laplacian);
[param.lambdaSym,indexSym] = sort(diag(param.eigenVal));
param.lambda_power_matrix(:,2) = param.lambdaSym;

for i = 1:max(param.K) + 1
    param.lambda_power_matrix(:,i) = param.lambda_power_matrix(:,2).^(i-1);
end

%% Initialize D:
[initial_dictionary, param] = construct_dict(param);

maxIter = 30;
X_norm_train = zeros(maxIter,1);
D_norm_train = zeros(maxIter,1);
norm_temp_W = zeros(maxIter,1);
D_diff = cell(maxIter,1);

for big_epoch = 1:maxIter

    param.iterN = big_epoch;

    if big_epoch == 1
        learned_dictionary = initial_dictionary;
        learned_W = initial_W;
        g_ker = zeros(param.N, param.S);
    end

    %---------optimise with respect to X---------%

    disp(['Epoch... ',num2str(big_epoch)]);
    x = OMP_non_normalized_atoms(learned_dictionary,param.y, param.T0);

    if mod(big_epoch,5) ~= 0
```

```
        %------optimize with respect to alpha------%

        [param,cpuTm] = coefficient_update_interior_point(Y,x,param,'sdpt3',g_ker);
%         cpuTime((big_epoch + 1)/2) = cpuTm;
    else
        %--------optimise with respect to W--------%
        disp('Graph learning step');
        maxEpoch = 1; %number of graph updating steps before updating sparse codes (x)↙
again
        beta = 10^(-2); %graph sparsity penalty
        old_L = param.Laplacian;
        [param.Laplacian, learned_W] = update_graph(x, alpha, beta, maxEpoch, param,↙
learned_W);
        alpha = alpha*0.985; %gradient descent decreasing
    end

    % Re-obtain D
    [learned_dictionary, param] = construct_dict(param);

    % Keep track of the evolution of D
    D_norm_train_def = 'norm(learned_dictionary - comp_D)';
    D_norm_train(big_epoch) = norm(learned_dictionary - comp_D);

    % Analyse the structural difference between learned Dictionaries
    D_diff{big_epoch} = (learned_dictionary - comp_D);

    % Keep track of the evolution of X and W
    X_norm_train(big_epoch) = norm(x - comp_train_X);
    norm_temp_W(big_epoch) = norm(learned_W - comp_W);
end

% % %         maxEpoch = 1; %number of graph updating steps before updating sparse↙
codes (x) again
% % %         beta = 10^(-2); %graph sparsity penalty
% % %         old_L = param.Laplacian;
% % %         [param.Laplacian, learned_W] = update_graph(x, alpha, beta, maxEpoch,↙
param, learned_W);
% % %         alpha = alpha*0.985; %gradient descent decreasing
% % %         [learned_dictionary, param] = construct_dict(param);

%% At the end of the cycle I have:
% param.alpha --> the original coefficients;
% X           --> the learned sparsity mx;
% learned_W   --> the learned W from the old D and alpha coeff;
% learned_dictionary --> the learned final dictionary;
% cpuTime     --> the final cpuTime

%constructed graph needs to be tresholded, otherwise it's too dense
%fix the number of desired edges here at nedges
nedges = 4*29;
final_Laplacian = treshold_by_edge_number(param.Laplacian, nedges);
final_W = learned_W.*(final_Laplacian~=0);


%% Estimate the final reproduction error
```

```
X_train = x;
x = OMP_non_normalized_atoms(learned_dictionary,TestSignal, param.T0);
errorTesting_Pol = sqrt(norm(TestSignal - learned_dictionary*x,'fro')^2/size↙
(TestSignal,2));
disp(['The total representation error of the testing signals is: ',num2str↙
(errorTesting_Pol)]);

%% Last eigenDecomposition, needed to compare the norm of the lambdas

[param.eigenMat, param.eigenVal] = eig(final_Laplacian);
[param.lambda_sym,index_sym] = sort(diag(param.eigenVal));

%% Compute the l-2 norms

X_norm_test = norm(x - comp_X);
total_X = [X_train x];
total_X_norm = norm(total_X - [comp_train_X comp_X]);
% X_norm_train = X_norm_train';
W_norm = norm(comp_W - learned_W); %Normal norm
W_norm_thr = norm(comp_W - final_W); %Normal norm of the thresholded adjacency matrix
norm_initial_W = norm(initial_W - comp_W);
norm_init_D = norm(initial_dictionary - comp_D);

%% Write down the definition of the norms for better clearance
norm_initial_W_def = 'norm(initial_W - comp_W)';
norm_temp_W_def = 'norm(learned_W - comp_W)';
X_norm_train_def = 'norm(X - comp_train_X)';
W_norm_thr_def = 'norm(comp_W - final_W) --> Normal norm of the thresholded adjacency↙
matrix';
W_norm_def = 'norm(comp_W - learned_W)';
total_X_norm_def = 'norm(total_X - [comp_train_X comp_X])';
X_norm_test_def = 'norm(X - comp_X)';
norm_init_D_def = 'norm(initial_dictionary - comp_D)';

%% Graphically represent the behavior od the learned entities

figure('name','Behavior of the X_norm_train (blue line) and the D_norm_train (orange↙
line)')
hold on
grid on
plot(1:maxIter,X_norm_train);
plot(1:maxIter,D_norm_train);
hold off

filename = [path,num2str(ds_name),'\behaviorX_','.png'];
saveas(gcf,filename);

%% Represent the kernels

param.lambda_power_matrix(:,2) = param.lambda_sym;
for i = 1:max(param.K) + 1
    param.lambda_power_matrix(:,i) = param.lambda_power_matrix(:,2).^(i-1);
end

for i = 1 : param.S
```

```
    for n = 1:param.N
        g_ker(n,i) = param.lambda_power_matrix(n,:)*param.alpha{i};
    end
end

figure('Name','Comparison between the Kernels')
subplot(1,2,1)
title('Original kernels');
hold on
for s = 1 : param.S
    plot(comp_lambdaSym,comp_ker(:,s));
end
hold off
subplot(1,2,2)
title('learned kernels');
hold on
for s = 1 : param.S
    plot(param.lambda_sym,g_ker(:,s));
end
hold off

%% Save results to file

% The kernels
filename = [path,num2str(ds_name),'\Comp_kernels_','.png'];
saveas(gcf,filename);

% The norms
norm_temp_W = norm_temp_W';
filename = [path,num2str(ds_name),'\Norms_',num2str(ds_name),'.mat'];
save↙
(filename,'W_norm_thr','W_norm','X_norm_train','norm_temp_W','X_norm_test','norm_initi↙
al_W','total_X_norm');

% The Output data
filename = [path,num2str(ds_name),'\Output_',num2str(ds_name),'.mat'];
learned_eigenVal = param.lambda_sym;
save↙
(filename,'ds','learned_dictionary','learned_W','final_W','Y','learned_eigenVal','erro↙
rTesting_Pol');

%% Verify the results with the precision recall function
learned_L = diag(sum(learned_W,2)) - learned_W;
learned_Laplacian = (diag(sum(learned_W,2)))^(-1/2)*learned_L*(diag(sum(learned_W,2)))↙
^(-1/2);

comp_L = diag(sum(comp_W,2)) - comp_W;
comp_Laplacian = (diag(sum(comp_W,2)))^(-1/2)*comp_L*(diag(sum(comp_L,2)))^(-1/2);

[optPrec, optRec, opt_Lapl] = precisionRecall(comp_Laplacian, learned_Laplacian);
filename = [path,num2str(ds_name),'\ouput_PrecisionRecall_',num2str(ds_name),'.mat'];
save(filename,'opt_Lapl','optPrec','optRec');
```