```
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\GeneratingKernels\Results'); %Folder↙
containing the heat kernel coefficietns
addpath('C:↙
\Users\Cristina\Documents\GitHub\OrganizedFiles\DataSets\Comparison_datasets\'); %↙
Folder containing the copmarison datasets
addpath('C:\Users\Cristina\Documents\GitHub\OrganizedFiles\DataSets\'); %Folder↙
containing the training and verification dataset

flag = 2;

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
        X = 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); % polynomial degree of each subdictionary
param.c = 1; % spectral control parameters
param.epsilon = 0.05;%0.02; % we assume that epsilon_1 = epsilon_2 = epsilon
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;
grad_desc = 2; %gradient descent parameter, it decreases with epochs
path = ['C:↙
\Users\Cristina\Documents\GitHub\OrganizedFiles\Graph&DictLearning\LearningG&D_byFunct↙
ions\Results\',num2str(ds_name),'\']; %Folder containing the results to save

%% Obtain the initial Laplacian and eigenValues for comparison
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
[out] = obtain_laplacian(comp_W,param,comp_alpha);
comp_Laplacian = out.Laplacian; comp_lambdaSym = out.lambdaSym;
comp_lambdaPowerMx = out.lambdaPowerMx; comp_ker = out.ker; comp_eigenMat = out.↙
eigenMat;

%% Initialise W and Laplacian

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

% % % if flag == 4
% % %     uniform_values = unifrnd(0,1,[1,param.N]);
% % %     sigma = 0.2;
% % %     [param.Laplacian,initial_W] = random_geometric(sigma,param.N,uniform_values,↙
0.6);
```

```
% % % else
% % %     [param.Laplacian,initial_W] = init_by_weight(param.N);
% % % end

[param.eigenMat, param.eigenVal] = eig(param.Laplacian);
[param.lambdaSym,indexSym] = sort(diag(param.eigenVal));
param.Laplacian_powers{1} = param.Laplacian;
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);
    param.Laplacian_powers{i} = param.Laplacian^(i-1);
end

%% Initialize alphas

for i = 1:param.S
    param.alpha(:,i) = comp_alpha((i-1)*(max(param.K)+1) + 1:(max(param.K)+1)*i);
end

%% Initialize D

[initial_dictionary, param] = construct_dict(param);
% % % [initial_dictionary(:,1 : param.J)] =  initialize_dictionary(param);

for big_epoch = 1:8

    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);

    % Keep track of the evolution of X
    X_norm_train(big_epoch) = norm(X - comp_train_X);

    if mod(big_epoch,9) == 0
                    %------optimize with respect to alpha------%

         [param,cpuTm] = coefficient_update_interior_point(Y,X,param,'sdpt3',g_ker);
%          cpuTime((big_epoch + 1)/2) = cpuTm;

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

    else
                    %--------optimise with respect to W--------%

        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, grad_desc, beta, maxEpoch,↙
param, learned_W);

        % Re obtain D
        [learned_dictionary, param] = construct_dict(param);
        grad_desc = grad_desc*0.985; %gradient descent decreasing

        % Keep track of the evolution of X
        norm_temp_W(big_epoch) = norm(learned_W - comp_W);
    end

    % 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);
end

%% At the end of the cycle I have:
% param.alpha --> the learned 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

%% 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)]);

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

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

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

%% 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

filename = [path,'Comp_kernels_','.png'];
saveas(gcf,filename);

%% Compute the l-2 norms

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

% 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_temp_X(big_epoch + 1) = norm(X - temp_X);
% % % W_norm_FRO = sqrt(norm(comp_W - learned_W,'fro')^2/size(comp_W,2)); %Frobenius↙
norm
% % % W_norm_thr_FRO = sqrt(norm(comp_W - final_W,'fro')^2/size(comp_W,2)); %Frobenius↙
norm of the thresholded adjacency matrix

%% 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(1:8,X_norm_train);
plot(1:8,D_norm_train);
hold off

filename = [path,'behaviorX_','.png'];
saveas(gcf,filename);
%% Save the results to file

% The norms
norm_temp_W = norm_temp_W';
X_norm_train = X_norm_train';
filename = [path,'\Norms_',num2str(ds_name),'.mat'];
save↙
(filename,'W_norm_thr','W_norm_thr_def','W_norm_def','W_norm','X_norm_train','X_norm_t↙
rain_def','norm_temp_W',...
↙
'norm_temp_W_def','X_norm_test','X_norm_test_def','norm_initial_W','norm_initial_W_def↙
',...
    'total_X_norm','total_X_norm_def','D_norm_train','D_norm_train_def','norm_init_D',↙
'norm_init_D_def');

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

%% Verify the results with the precision recall function
learned_Laplacian = final_Laplacian;
[optPrec, optRec, opt_Lapl] = precisionRecall(comp_Laplacian, learned_Laplacian);
filename = [path,'\ouput_PrecisionRecall_',num2str(ds_name),'.mat'];
save(filename,'opt_Lapl','optPrec','optRec');
```