Machine Learning

Homework 7

Report

Name: Sergiy Shelekh
ID: 0760822
Email: proxitrone@gmail.com

Introduction

The first part of this homework is to use Principal Component Analysis (PCA) and Linear Discriminative Analysis (LDA) on the Yale database of facial expression images to find eigenfaces and fisherfaces of those images, be able to classify them. In the second part we are given an implementation of t-SNE embedding technique, which we need to convert to symmetric SNE, visualize the operation of both, their corresponding high- and low-dimension distributions of pairwise similarities, and look at whether different choices of perplexity will give us different final embeddings. In this assignment I use MATLAB programming language, as it allows fast and efficient computations of things like Gram matrix and has good debugging capabilities.

PCA: Implementation

The first thing that we need to do for this part is some data preprocessing. We are given the Yale database with images, but we need to load them into our program for future use first. Following the referenced paper, we crop and downsample the images.

```
 6      % Load Yale database
 7 -    root = [];
 8 -    target_rescale = [41, 29];
 9 -    crop_param = [47, 60, 100, 150];
10 -    [Train, Test] = load_yale(root, crop_param, target_rescale);
```

To load all images into matrices right away we take the property of the directory, where they are stored to get the number of images to load. Images are loaded to a single matrix (Train or Test) as column vectors, each column is a separate observation.

```
function [Train,Test] = load_yale(root, crop_param, target_rescale)
%LOAD_YALE Load the Yale_Face_Database into matrices in matlab
%   Train and Test sets are cells with matrices for each test_subject (class)
%   Perform normalization and resizing D
    face_vec_dim = target_rescale(1)*target_rescale(2);

    train_path = [root ,'Yale_Face_Database\Training\'];
    test_path = [root ,'Yale_Face_Database\Testing\'];

    D_train = dir(fullfile(train_path, '*.pgm'));
    D_test = dir(fullfile(test_path, '*.pgm'));

    Train = zeros(face_vec_dim, numel(D_train));
    Test = zeros(face_vec_dim, numel(D_test));
```

```
    %% Load the train set
    for k=1:numel(D_train)
        face = imread(fullfile(train_path, D_train(k).name));
        face = preprocess_face(face, crop_param, target_rescale);
        Train(:, k) = face(:);
    end

    %% Load the test set
    for k=1:numel(D_test)
        face = imread(fullfile(test_path, D_test(k).name));
        face = preprocess_face(face, crop_param, target_rescale);
        Test(:, k) = face(:);
    end
```

Then, some more processing on data is done, we divide datapoints in both training and testing set into separate classes, this will come in handy during LDA and for classification.

```
%% Divide Training and Testing set by classes (every 9 datapoints belong to 1 subject)
classes = 15;
Train_classes = cell(classes, 1);
Test_classes = cell(classes, 1);
for c=1:classes
    Train_classes{c} = zeros(size(Train,1), 9);
    for i=1:9
        Train_classes{c}(:,i) = Train(:, (c-1)*9+i);
    end

    Test_classes{c} = zeros(size(Test,1), 2);
    for i=1:2
        Test_classes{c}(:,i) = Test(:, (c-1)*2+i);
    end
end
```

After all preprocessing is done, we can go to the PCA part.

```
30      %% Perform PCA
31 -    W_pca = myPCA(Train, k);
```

PCA procedure is quite straightforward, we first zero-mean our data, and then compute it's covariance:

```
1      function [W_pca] = myPCA(data, k)
2      %MYPCA Use PCA to show the first 25 eigenfaces and reconstruction
3      %   Compute the Covariance matrix and take the first k eigenvectors
4 -        data_mean = mean(data, 2);
5 -        data = data-data_mean;
6 -        C = cov(data');
```

This covariance matrix or total scatter plays significant role in PCA, since a number of first largest eigenvectors of this covariance gives us exactly the orthogonal projection weights we need.

```
7 -        [eigVec, eigVal] = eig(C);
8          % sort eigenvectors and corresponding eigenvalues
9 -        [d, ind] = sort(diag(eigVal), 'descend');
10 -       eigVal = eigVal(:, ind);
11 -       eigVec = eigVec(:, ind);
12
13 -       W = eigVec(:,1:k);
14 -       W_pca = eigVec;
```

We then show the resulting eigenfaces

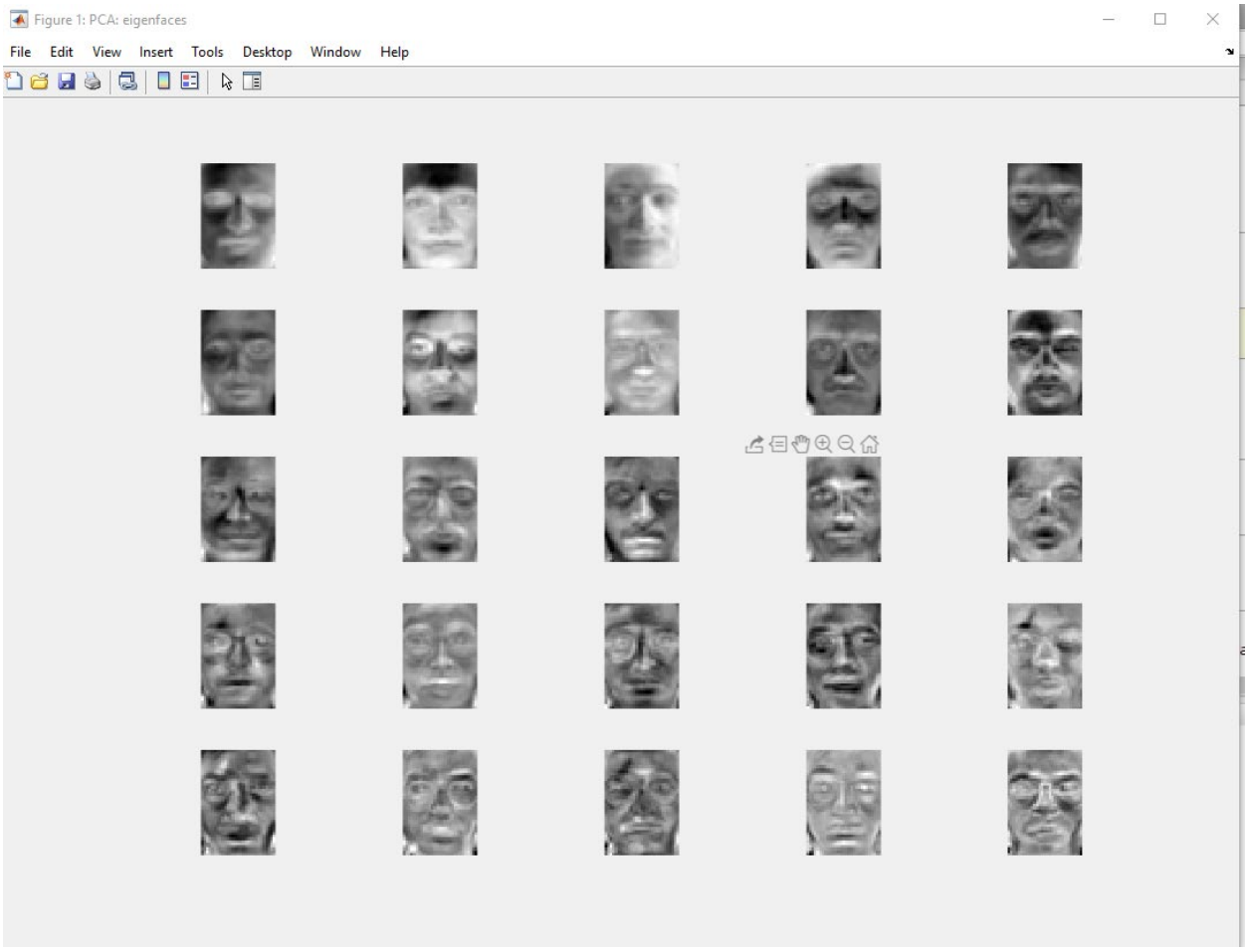
```
15         %Get data after projection
16 -       z=data'*W;
17
18         % Display the eigenfaces
19 -       figure('Name', 'PCA: eigenfaces');
20 -       for i=1:k
21 -           A = reshape(W(:, i), [41, 29]);
22 -           subplot(k/5, 5, i);
23             %A = abs(A)./max(abs(A));
24 -           A = A+abs(min(A, [], 'all'));
25 -           A = A./max(A, [], 'all');
26 -           imshow(A);
27 -       end
```

And 10 random reconstructed images:

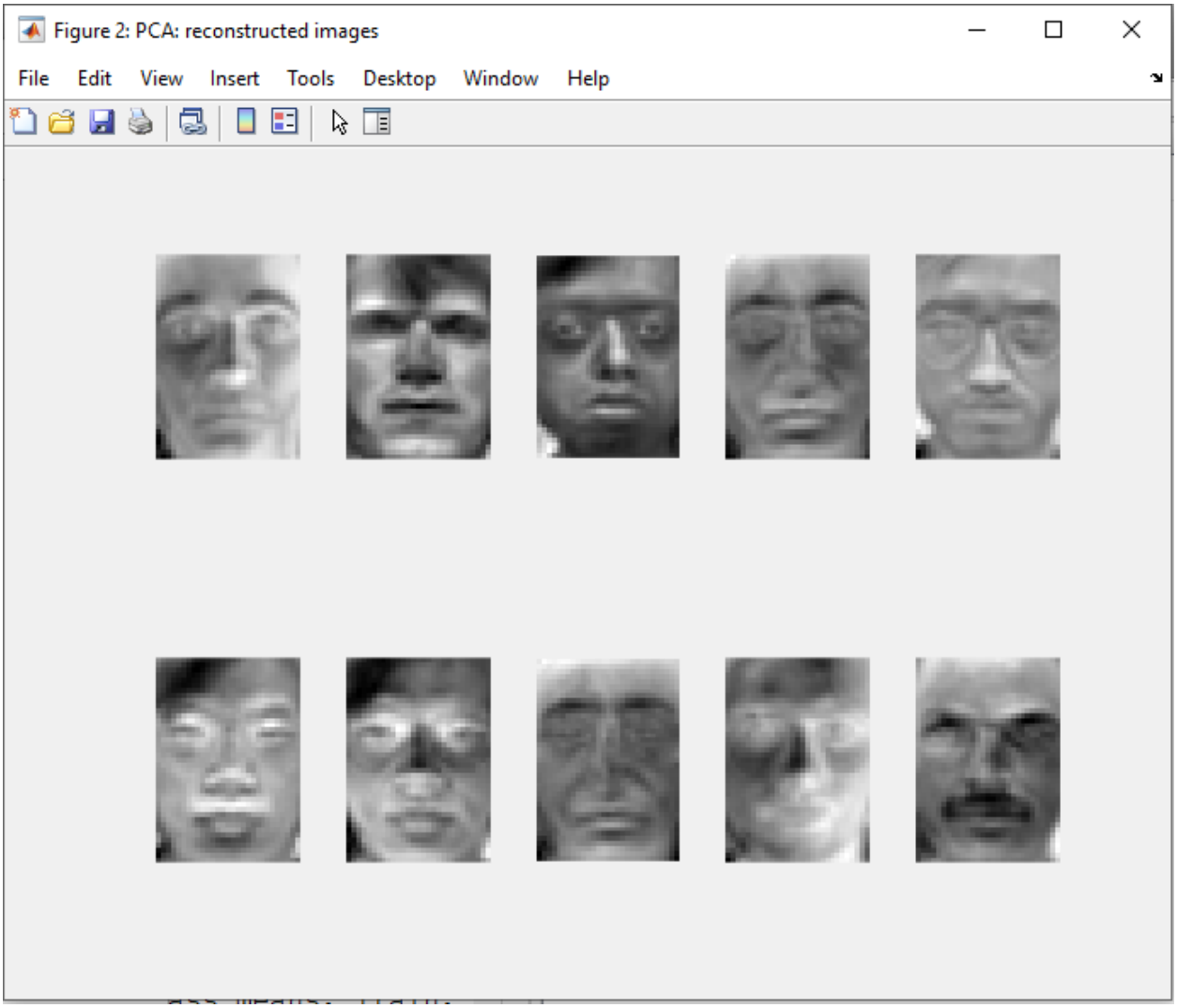
```
28          % Build the reconstructed images
29 -        reconstructed = W*z';
30          % Show 10 random reconstructed
31 -        figure('Name', 'PCA: reconstructed images');
32 -        for i=1:10
33 -            k=uint8(rand*134+1);
34 -            A = reshape(reconstructed(:, k), [41, 29]);
35 -            subplot(2, 5, i);
36 -            A = A+abs(min(A, [], 'all'));
37 -            A = A./max(A, [], 'all');
38 -            imshow(A);
39 -        end
40 -    end
```

PCA: Results

We show 25 eigenfaces:



And 10 random reconstructed images:



This approach gives us some nice results, eliminating varying lighting conditions present in the original dataset

LDA: Implementation

We move on to perform LDA, where in order to get fisherfaces, we'll need to use results obtained from PCA.

```
33
34    %% Perform LDA
      [W_lda, class_means] = myLDA(Train_classes, 10, Train, W_pca);
```

LDA is a kind of supervised algorithm, where we want to project our high-dimensional data into some low dimension, while maximizing between-class scatter and minimizing the within-class scatter. Both these values can be easily computed:

$$\textit{within-class scatter: } S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$$

$$\text{and } \mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$$

$$\textit{between-class scatter:}$$

$$S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

$$\text{where } \mathbf{m} = \frac{1}{n} \sum x$$

And that's exactly what we are doing next

```
function [W_opt, means] = myLDA(classes_data, g, reg_data, W_pca)
%MYLDA Perform LDA on Yale dataset
%    Compute class centers first
    classes = size(classes_data, 1);
    % Compute class means
    means = zeros(size(classes_data{1},1),classes);
    for c=1:classes
        means(:, c) = mean(classes_data{c}, 2);
    end
```

First, compute class means and them display them for our reference

```
    % Show class means
    figure('Name','LDA: class means');
    for c=1:classes
        A = reshape(means(:, c), [41, 29]);
        subplot(3, 5, c);
        imshow(A);
    end
```

We then proceed to computing all data mean and use it to zero-mean our data

```
19        % Compute all data mean
20 -      data_mean = mean(means, 2);
21 -      reg_data = reg_data-data_mean;
```

Now we can start getting those within- and between-class scatter matrices:

```
22        % Compute between-class scatter
23 -      S_b = 0;
24 -      for c=1:classes
25 -          class_elem_num = size(classes_data{c},2);
26 -          diff = means(:,c)-data_mean;
27 -          S_b = S_b + class_elem_num*(diff*diff');
28 -      end
```

```
30        % Compute within-class scatter
31 -      S_w = 0;
32 -      for c=1:classes
33 -          class_elem_num = size(classes_data{c},2);
34 -          for i=1:class_elem_num
35 -              diff = (classes_data{c}(:,i)-means(:,c))-means(:,c);
36 -              S_w = S_w + (diff*diff');
37 -          end
38 -      end
```

Originally, we would simply take first largest eigenvalues of $S_W^{-1}S_B$ as our projection weight matrix, but in our case S_W is singular, which makes the inverse undesirable. So, we use the result from PCA to overcome that singularity problem. Project both within- and between-class scatters by the PCA projection and only then take the inverse:

```
39        % Get the largest eigenvalues
40 -      W_pca = W_pca(:,1:14);
41
42 -      S_bb = W_pca'*S_b*W_pca;
43 -      S_ww = W_pca'*S_w*W_pca;
44        % Compute S_w^-1*S_b and take first q largest eigenvectors as W
45 -      Tmp = S_ww\S_bb;
```

Take the largest eigenvalues as the Fisher's linear decomposition weights:

```
47 -        [eigVec, eigVal] = eig(Tmp);
48          % sort eigenvectors and corresponding eigenvalues
49 -        [d, ind] = sort(diag(eigVal), 'ascend');
50 -        eigVal = eigVal(:, ind);
51 -        eigVec = eigVec(:, ind);
52
53      %      W_fld = eigVec(:,1:q);
54 -        W_fld = eigVec;
```

And then combine this projection with PCA to get optimal weights and our fisherfaces:

```
55          % Fisherfaces
56 -        W_opt = W_fld'*W_pca';
57 -        W_opt = W_opt';
```

Show them first

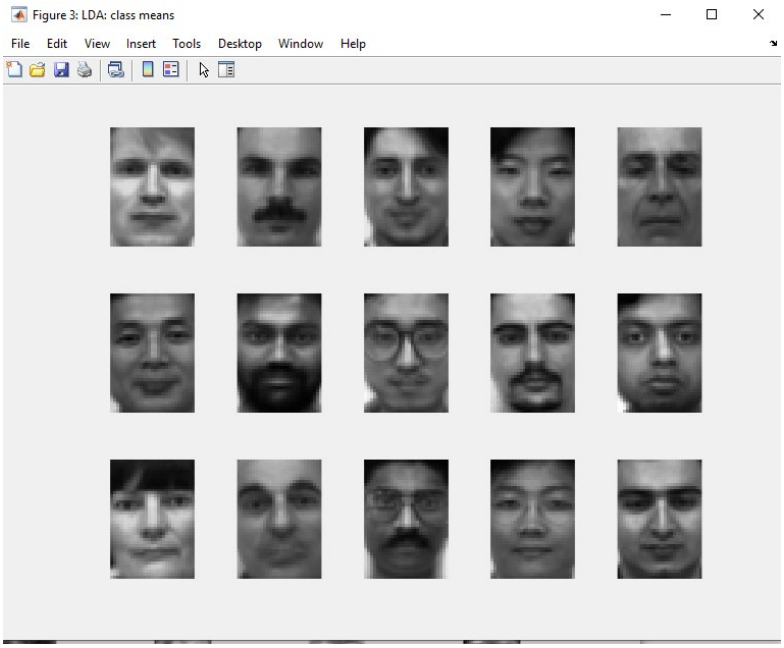
```
58          % Display the fisherfaces
59 -        figure('Name', 'LDA: fisherfaces');
60 -        for i=1:size(W_opt, 2)
61 -            A = reshape(W_opt(:, i), [41, 29]);
62 -            subplot(5, 5, i);
63              %A = abs(A)./max(abs(A));
64 -            A = A+abs(min(A, [], 'all'));
65 -            A = A./max(A, [], 'all');
66 -            imshow(A);
67 -        end
```

And then perform reconstruction:

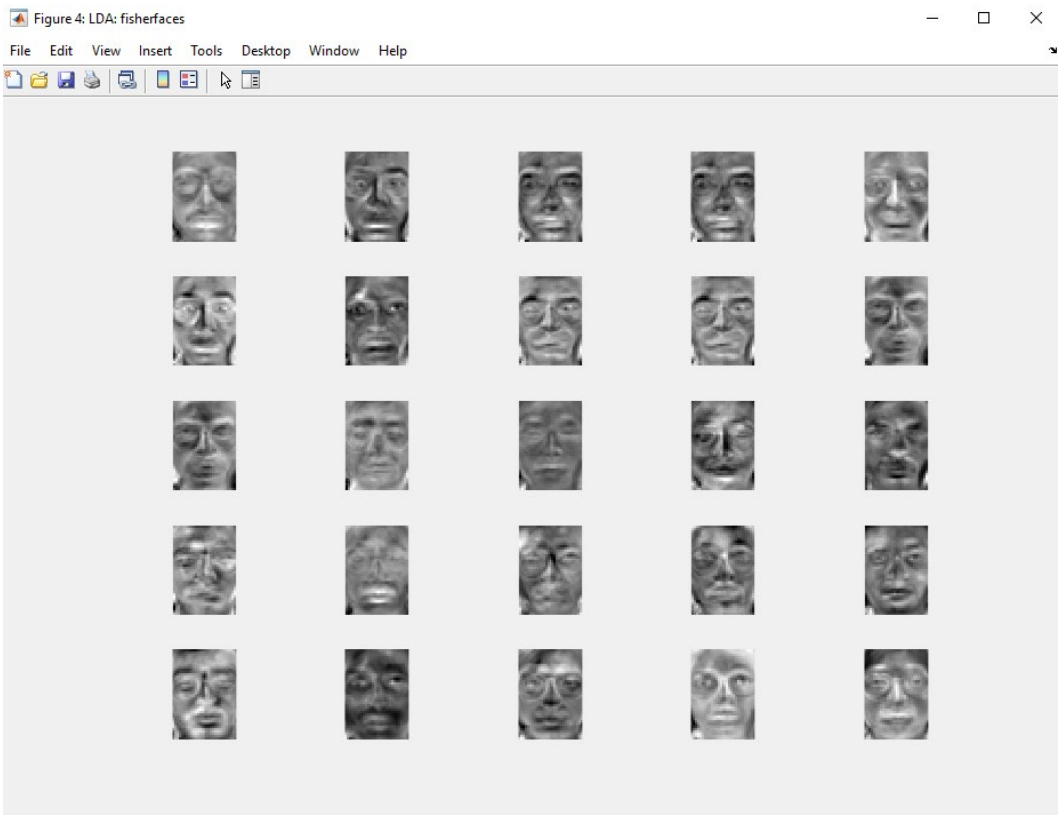
```
69          % Get data after projection
70 -        z = reg_data'*W_opt;
71
72 -        reconstructed = W_opt*z';
73          % Show 10 random reconstructed
74 -        figure('Name', 'LDA: reconstructed images');
75 -        for i=1:10
76 -            k=uint8(rand*134+1);
77 -            A = reshape(reconstructed(:, k), [41, 29]);
78 -            subplot(2, 5, i);
79 -            A = A+abs(min(A, [], 'all'));
80 -            A = A./max(A, [], 'all');
81 -            imshow(A);
82 -        end
```

LDA: Results

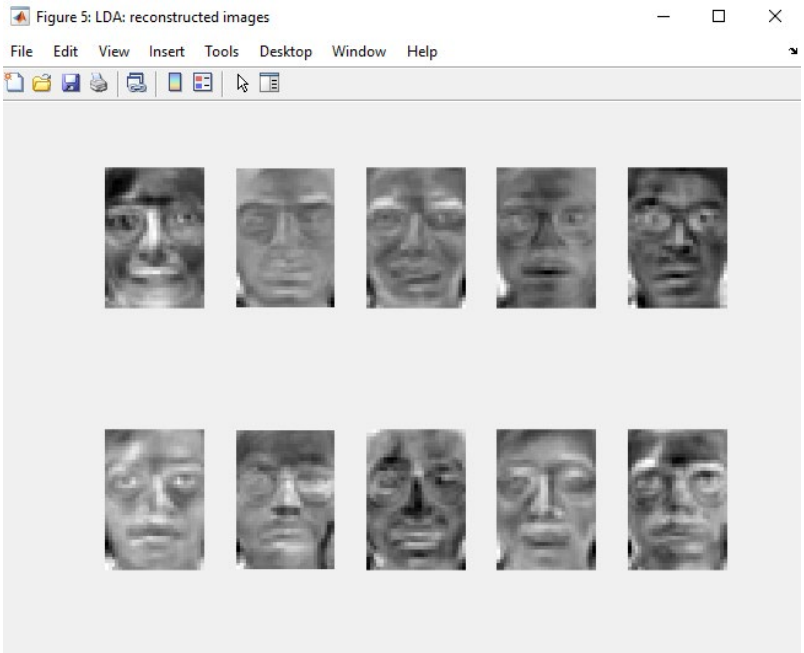
Class means:



Fisherfaces:



Reconstructed images:



PCA&LDA Classification: Implementation

We now have both projection matrices from PCA and LDA, so we can go on to classifying images from our testing set

```
35      %% Classify regular PCA and LDA
36 -    [acc_pca, acc_lda] = classify_pca_lda(W_pca, W_lda, Test_classes, class_means);
37 -    disp(['Pca acc: ', num2str(acc_pca)]);
38 -    disp(['Lda acc: ', num2str(acc_lda)]);
```

In this classification we project our testing datapoint onto the low-dimensional spaces, given by PCA and LDA, and compute the distance between this projection and projected class means.

```
1       function [acc_pca, acc_lda] = classify_pca_lda(W_pca, W_lda, classes_test_data, means)
2       %CLASSIFY_PCA_LDA Classify images from test set using pca and lda
3       %   To classify test images, we need to project each image using
4       %   corresponding transformation, project class means, compute distance of
5       %   our test projection to each of the projected means, minimum distance
6       %   mean is going to be our class
7 -         W_pca = W_pca(:, 1:14);
```

Choose first largest components, project class means:

```
9 -         classes_num = size(classes_test_data, 1);
10 -        proj_means_pca = W_pca'*means;
11 -        proj_means_lda = W_lda'*means;
12 -        acc_pca = 0;
13 -        acc_lda = 0;
14 -        num_data = 0;
```

Start iterating through the testing set, project each point and compute it's distance to every mean, choose the lowest one to be our class.

```
    for c=1:classes_num
        class_elem_num = size(classes_test_data{c},2);
        for i=1:class_elem_num
            sample_proj_pca = W_pca'*classes_test_data{c}(:,i);
            sample_proj_lda = W_lda'*classes_test_data{c}(:,i);
            distance_pca = zeros(1, classes_num);
            distance_lda = zeros(1, classes_num);
            for k=1:classes_num
                distance_pca(1, k) = norm(sample_proj_pca-proj_means_pca(:,k))^2;
                distance_lda(1, k) = norm(sample_proj_lda-proj_means_lda(:,k))^2;
            end
```

Find the minimum and compare it to the true label, compute accuracy

```
            [val, ind_pca] = min(distance_pca);
            if ind_pca == c
                acc_pca = acc_pca +1;
            end
            [val, ind_lda] = min(distance_lda);
            if ind_lda == c
                acc_lda = acc_lda +1;
            end
            num_data = num_data+1;
        end
    end

    acc_pca = acc_pca/num_data;
    acc_lda = acc_lda/num_data;
```

PCA&LDA Classification: Results

Resulting accuracies are quite high:

```
Pca acc: 0.83333
Lda acc: 0.93333
```

LDA performs better than PCA, which is expected, since it's a more accurate supervised technique

Kernel PCA&LDA: Implementation

After we have finished with the regular PCA and LDA, where we only use linear distances as our datapoint correlation, we move on to a more elaborate version of the above two algorithms, where we, once again, use the kernel trick to allow the boundaries to be nonlinear. I use two different kernels to compare: quadratic polynomial kernel and RBF kernel.

```
%% Perform Gauss Kernel PCA
W_gauss_pca = myKernelPCA(Train, k, 'poly');
```

Kernel PCA is done quite similar to regular PCA, but instead of the regular covariance matrix, we use pairwise distances between points in kernel space

```
function [W_pca] = myKernelPCA(data, k, kernel_type)
%MYKERNELPCA Perform Kernel PCA
%   Same as PCA, but now instead of covariance matrix we use kernel matrix

    data_mean = mean(data, 2);
    data = data-data_mean;
    C = compute_kernel(data, data, kernel_type);
    [eigVec, eigVal] = eig(C);
    % sort eigenvectors and corresponding eigenvalues
    [d, ind] = sort(diag(eigVal), 'descend');
    eigVal = eigVal(:, ind);
    eigVec = eigVec(:, ind);

    W = eigVec(:,1:k);
    W_pca = eigVec;
```

We compute all kernels in a similar fashion as in previous assignments

```
function [Gram] = compute_kernel(dataA, dataB, kernel_type)
%COMPUTE_KERNEL Compute the gram matrix
%   Different kernels available: rbf, gauss, 2nd order polynomial
    gamma_s = 1/100;

    K = pdist2(dataA', dataB', 'euclidean');

    if strcmp(kernel_type, 'rbf')
        Gram = exp(-gamma_s*K);
    end

    if strcmp(kernel_type, 'gauss')
        Gram = (gamma_s/pi)*exp(-gamma_s*K);
    end

    if strcmp(kernel_type, 'poly')
        Gram = (dataA'*dataB+gamma_s).^2;
    end


end
```

Implementation of kernel LDA is a bit more interesting. We now work in the feature space, where we also need to find means of our classes and distances between points. To simplify computations, we take advantage of matrix operations and we need a class matrix, where rows are observations and columns are classes, it has a 1 in a corresponding place if a datapoint belongs to that class.

```
function [W_opt, means] = myKernelLDA(data, k, kernel_type)
%MYKERNELLDA Perform Kernel LDA
%   Compute within-class and between-class scatter using a kernel
% Need to center data in kernel space

    obs_num = size(data, 2);
    class_num = 15;
    % Generate class vectors
    class_vec = zeros(obs_num, class_num);
    for c=1:class_num
        sample_num = 9;
        for i=1:sample_num
            class_vec((c-1)*sample_num + i, c) = 1;
        end
    end
```

We then compute the Gram matrix and regularize it

```
    % Compute the gram matrix
    K = compute_kernel(data, data, kernel_type);
    K = K./obs_num;
```

Find the means of our classes in feature space

```
    %Compute dual representation of averages (in kernel space)
    means = zeros(obs_num, class_num);
    data_mean = zeros(obs_num, 1);

    class_obs = 11;
    for c=1:class_num
        means(:, c) = K* class_vec(:, c) ./class_obs;
        data_mean = data_mean + means(:, c);
    end
    data_mean = data_mean./class_num;
```

Now we have everything we need to compute within- and between-class scatter in the feature space. Notice the extensive use of our Gram matrix, no additional computations

```
32        % Compute the within-class and between-class scatter
33 -      M = zeros(obs_num, obs_num);
34 -      N = K*K';
35
36 -      for c = 1:class_num
37 -          M = M+(means(:, c)-data_mean)*(means(:, c)-data_mean)';
38 -          N = N-(sample_num*(means(:, c)*means(:, c)'));
39 -      end
40 -      M= M.*(class_num-1); %between-class unbiased scatter
41
42       %Regularizing
43 -      mK = abs(mean(K(:)));
44 -      C = 0.25*mK;
45 -      N = N+C*K;
```

After regularizing the within-class scatter, we solve the generalized eigenvector problem with the two different scatters we obtained:

```
47       % Extract eigenvalues and eigenvectors
48 -      [Vtmp, lambda] = eig(M, N);
49 -      lambda = real(diag(lambda));
50
51       % Sort the eigenvalues.
52 -      [~, index] = sort(abs(lambda), 'descend');
53 -      W_opt = Vtmp(:, index);
54
55 -      z = K*W_opt;
```

And, as before, our projection weights are given by the eigenvectors and we can go on to classification.

```
45    %% Classify Gauss kernel PCA and LDA
46 -  [acc_pca, acc_lda] = classify_kernel_pca_lda(W_gauss_pca, W_gauss_lda, Test_classes, class_means, Train, Test, 'poly');
47 -  disp(['poly pca acc: ', num2str(acc_pca)]);
48 -  disp(['poly lda acc: ', num2str(acc_lda)]);
```

Since we are now working in the feature space, we can't simply project class means onto the low dimension, we need to compute the kernel between testing and training points, project those points onto a low-d space, and compare distances there.

```
1     function [acc_pca, acc_lda] = classify_kernel_pca_lda(W_pca, W_lda, classes_test_data, means, data, test_data, kernel_type)
2     %CLASSIFY_KERNEL_PCA_LDA Perform classification using kernel PCA and LDA
3     %   Compute kernel distance between testpoints and trainpoints in feature
4     %   space, shortest distance is our class
5
6 -       classes_num = size(classes_test_data, 1);
7 -       num_train_data = size(data, 2);
8 -       num_test_data = size(test_data, 2);
9
10 -      acc_pca = 0;
11 -      acc_lda = 0;
12        %Compute distances between points and means in feature space
13 -      W_pca = W_pca(:, 1:14);
14 -      W_lda = W_lda(:, 1:14);
```

```
% Compute the test kernel matrix
K1 = compute_kernel(data, data, kernel_type);
K2 = compute_kernel(data, test_data, kernel_type);

K2 = K2./num_train_data;
K1 = K1./num_train_data;
```

Project data onto the discriminant axes:

```
% Project data onto the discriminant axes
proj_test_pca = K2'*W_pca;
proj_test_lda = K2'*W_lda;

proj_train_pca = K1'*W_pca;
proj_train_lda = K1'*W_lda;
```

Compute distances in the projected space:

```
% Compute distances in the projected space
distances_pca = pdist2(proj_test_pca, proj_train_pca, 'euclidean');
distances_lda = pdist2(proj_test_lda, proj_train_lda, 'euclidean');
```

Find which distances are the shortest

```
% Find the elements with shortest distances
[shortest, ind_pca] = min(distances_pca, [], 2);
[shortest, ind_lda] = min(distances_lda, [], 2);
```

Generate class vectors to compare classifications:

```
% Generate class vectors
class_vec_train = zeros(num_train_data, classes_num);
class_vec_test = zeros(num_test_data, classes_num);
for c=1:classes_num
    sample_num_train = 9;
    for i=1:sample_num_train
        class_vec_train((c-1)*sample_num_train + i, c) = 1;
    end

    sample_num_test = 2;
    for i=1:sample_num_test
        class_vec_test((c-1)*sample_num_test + i, c) = 1;
    end
end
```

And finally classify the points and update accuracies:

```
    % Classify test data point according to their closest neighbor
    for i=1:num_test_data
        if sum(class_vec_test(i, :) == class_vec_train(ind_pca(i), :)) == classes_num
%         [val, target_class] = max(class_vec_train(ind_pca(i), :));
            acc_pca = acc_pca+1;
        end

        if sum(class_vec_test(i, :) == class_vec_train(ind_lda(i), :)) == classes_num
%         [val, target_class] = max(class_vec_train(ind_pca(i), :));
            acc_lda = acc_lda+1;
        end
    end

    acc_pca = acc_pca/num_test_data;
    acc_lda = acc_lda/num_test_data;
end
```

Kernel PCA&LDA: Results

The results are as following:

```
poly pca acc: 0.66667
poly lda acc: 0.93333
RBF pca acc: 0.73333
RBF lda acc: 0.8
```

We see, that for polynomial kernel the accuracy decreased for PCA compared to regular version, which is most likely due to the inaccurate choice of hyperparameters. LDA shows same results in case of polynomial kernel, but lower accuracy for RBF. PCA rbf kernel performs better than polynomial

t-SNE and Symmetric SNE: Implementation

In this part we are given a code of implementation of t-SNE embedding technique, which we need to modify back to be a symmetric SNE. Main difference between the two, is the type of distribution used to measure spread of points in low-dimensional space. In case of symmetric SNE we use a gaussian distribution:

$$q_{ij} = \frac{\exp(- \mid\mid y_i - y_j \mid\mid^2)}{\sum_{k \neq l} \exp(- \mid\mid y_l - y_k \mid\mid^2)}$$

While in t-SNE we modify that distribution to be of t-distribution, which allows points with low probability to be farther away from each other in that low-dimensional space:

$$q_{ij} = \frac{(1+ \mid\mid y_i - y_j \mid\mid^2)^{-1}}{\sum_{k \neq l} (1+ \mid\mid y_i - y_j \mid\mid^2)^{-1}}$$

The next difference is the gradient of the objective function we want to minimize (KL divergence).

Objective:

$$C = KL(P||Q) = \sum_i \sum_{j \neq i} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

In case of symmetric SNE, the gradient is given by:

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

While for t-SNE we have:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + || y_i - y_j ||^2)^{-1}$$

Now we go on to identify and change the two corresponding parts in the given source code. I won't go into the detail of implementation of the given code, but generally, the program first normalizes the input and computes pairwise distances between points, to be further used in computing joint probabilities for affinity matrix in higher dimension.

```
76        % Compute pairwise distance matrix
77 -      sum_X = sum(X .^ 2, 2);
78 -      D = bsxfun(@plus, sum_X, bsxfun(@plus, sum_X', -2 * (X * X')));
79
80        % Compute joint probabilities
81 -      P = d2p(D, perplexity, 1e-5);% compute affinities using fixed perplexity
82 -      clear D
83
84        % Run t-SNE
85 -      if initial_solution
86 -          ydata = tsne_p(P, labels, ydata, mode, filename);
87 -      else
88 -          [ydata, P, Q] = tsne_p(P, labels, no_dims, mode, filename);
89 -      end
```

After we have the constant affinity matrix P, we go on to compute the low-dimensional affinity matrix Q, gradient of KL divergence w.r.t Q, and do a gradient step to eventually converge to a local optimal solution

```
63 -    for iter=1:max_iter
64
65          % Compute joint probability that point i and j are neighbors
66 -        sum_ydata = sum(ydata .^ 2, 2);
67 -        if mode == 0
68              % Student-t distribution
69 -            num = 1 ./ (1 + bsxfun(@plus, sum_ydata, bsxfun(@plus, sum_ydata', -2 * (ydata * ydata'))));
70 -        elseif mode == 1
71              % Normal distribution
72 -            num =  exp(-(bsxfun(@plus, sum_ydata, bsxfun(@plus, sum_ydata', -2 * (ydata * ydata')))));
73 -        end
74 -        num(1:n+1:end) = 0;                                 % set diagonal to zero
75 -        Q = max(num ./ sum(num(:)), realmin);               % normalize to get probabilities
```

Here we can see the computation of matrix Q for different distributions, student t-distribution is given by the authors, and normal distribution is our modification. Computation is pretty straightforward

```
77 -            if mode == 0
78                  % Compute the gradients (faster implementation) t-SNE gradient
79 -                L = (P - Q) .* num;
80 -                y_grads = 4 * (diag(sum(L, 1)) - L) * ydata;
81 -            elseif mode == 1
82                  % Compute the gradients (faster implementation) Normal gradient
83 -                L = (P - Q);
84 -                y_grads = 2 * (diag(sum(L, 1)) - L) * ydata;
85 -            end
```

This part is responsible for computing the gradient of KL divergence w.r.t matrix Q. Our modification is in computing the Normal gradient. Later in the program, the solution (ydata) is updated using the above gradient. The progress is visualized every 10 iterations and we also modify the visualization to save GIF images of the training process.

```
110             % Display scatter plot (maximally first three dimensions)
111 -           if ~rem(iter, 10) && ~isempty(labels)
112 -               if no_dims == 1
113 -                   scatter(ydata, ydata, 9, labels, 'filled');
114 -               elseif no_dims == 2
115 -                   scatter(ydata(:,1), ydata(:,2), 9, labels, 'filled');
116 -               else
117 -                   scatter3(ydata(:,1), ydata(:,2), ydata(:,3), 40, labels, 'filled');
118 -               end
119 -               axis tight
120 -               axis off
121 -               drawnow
```

```
123 -               frame = getframe(h);
124 -               im = frame2im(frame);
125 -               [imind,cm] = rgb2ind(im,256);
126                 % Write to the GIF File
127 -               if iter == 10
128 -                   imwrite(imind,cm,[filename, '.gif'], 'gif', 'Loopcount',inf, 'DelayTime',1);
129 -               else
130 -                   imwrite(imind,cm,[filename, '.gif'], 'gif','WriteMode','append', 'DelayTime',0.5);
131 -               end
132 -           end
133 -   end
```

We do the optimization for a total number of 1000 iterations and then output distribution pairwise similarities in both high- and low-dimensional spaces to visualize the result in the end.

```
perplexity = [30, 15, 60];
for i=1:3
    %% t-SNE
    [ydata_t, P_t, Q_t] = tsne(mnist2500_X, mnist2500_labels, 0, 2, 30, perplexity(i));
    figure('Name', ['tSNE P Distribution Visualization, perplexity', num2str(perplexity(i))]);
    A = (P_t-mean(P_t(:)));
    A = A+min(abs(A(:)));
    A = A./max(A(:));
    imshow(A);
    figure('Name', ['tSNE Q Distribution Visualization, perplexity', num2str(perplexity(i))]);
    A = (Q_t-mean(Q_t(:)));
    A = A+min(abs(A(:)));
    A = A./max(A(:));
    imshow(A);
```

As you can see, we first need to normalize those distributions for visualization and we do so for 3 different types of perplexity: 15, 30, and 60

```
    %% Symmetric SNE
    [ydata_n, P_n, Q_n] = tsne(mnist2500_X, mnist2500_labels, 1, 2, 30, perplexity(i));
    figure('Name', ['SymSNE P Distribution Visualization, perplexity', num2str(perplexity(i))]);
    A = (P_n-mean(P_n(:)));
    A = A+min(abs(A(:)));
    A = A./max(A(:));
    imshow(A);
    figure('Name', ['SymSNE Q Distribution Visualization, perplexity', num2str(perplexity(i))]);
    A = (Q_n-mean(Q_n(:)));
    A = A+min(abs(A(:)));
    A = A./max(A(:));
    imshow(A);
end
```

t-SNE and Symmetric SNE: Results

All GIF images are available in the root of the homework folder and are made during the program operation.

Perplexity 30:

t-SNE:

Final embedding

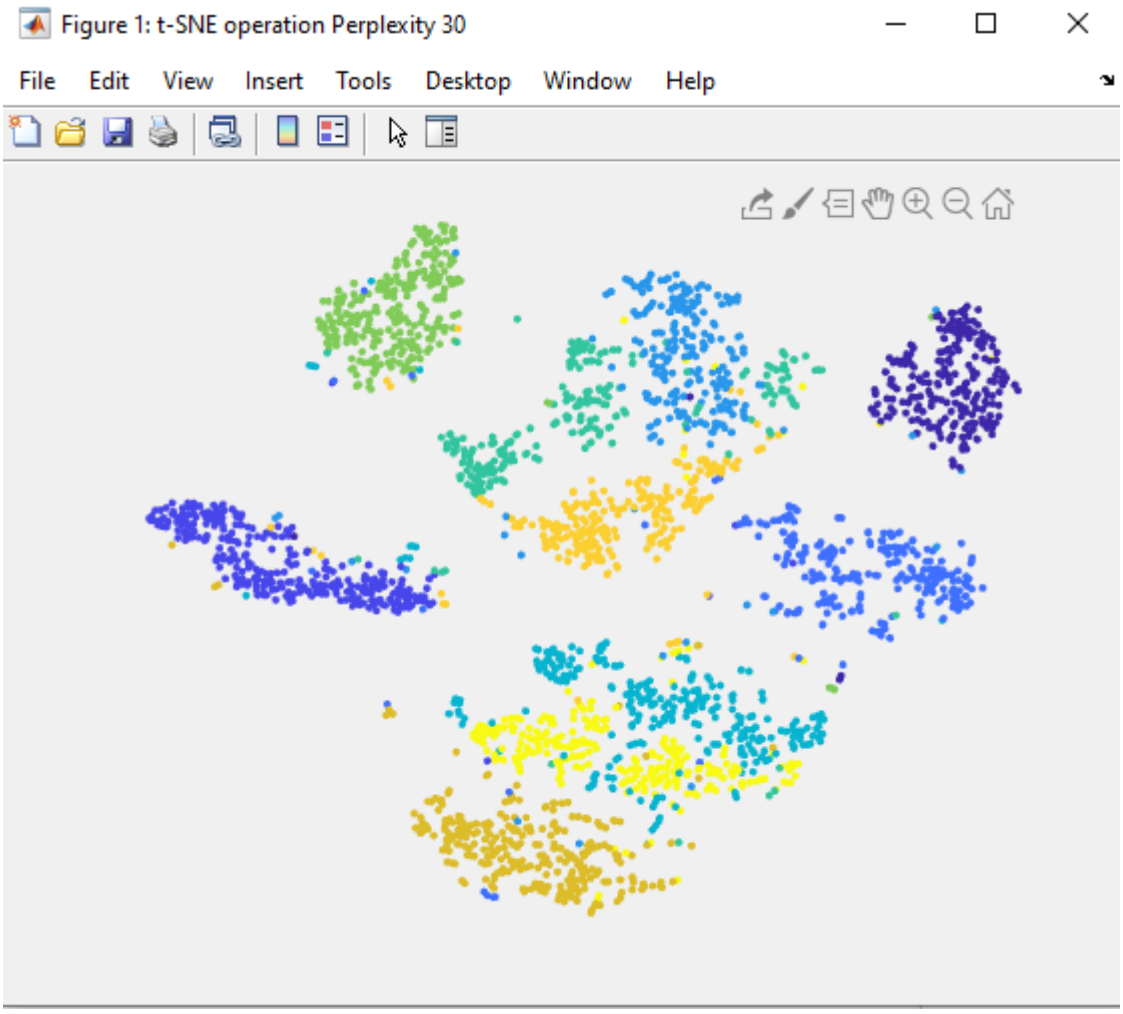
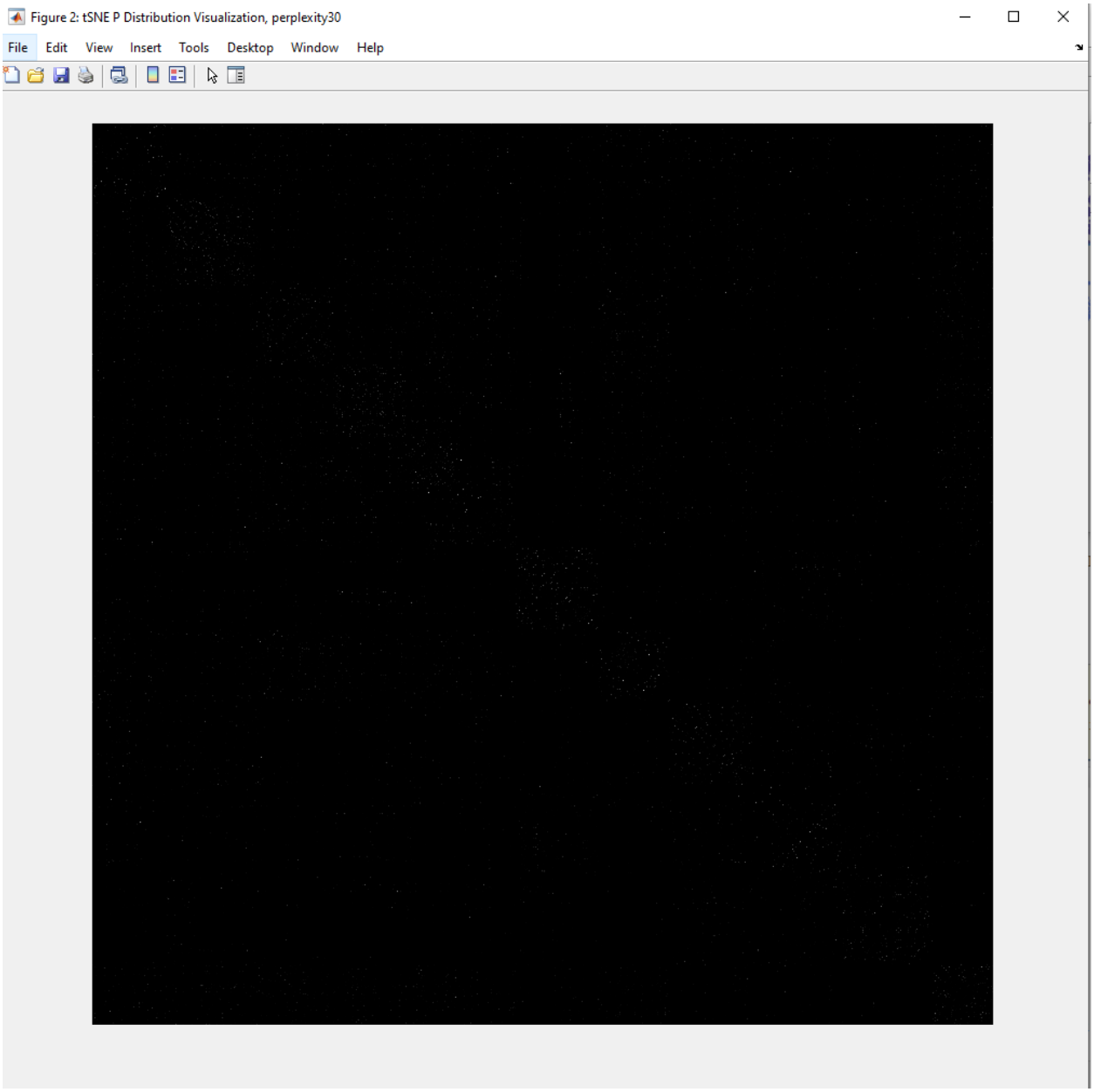
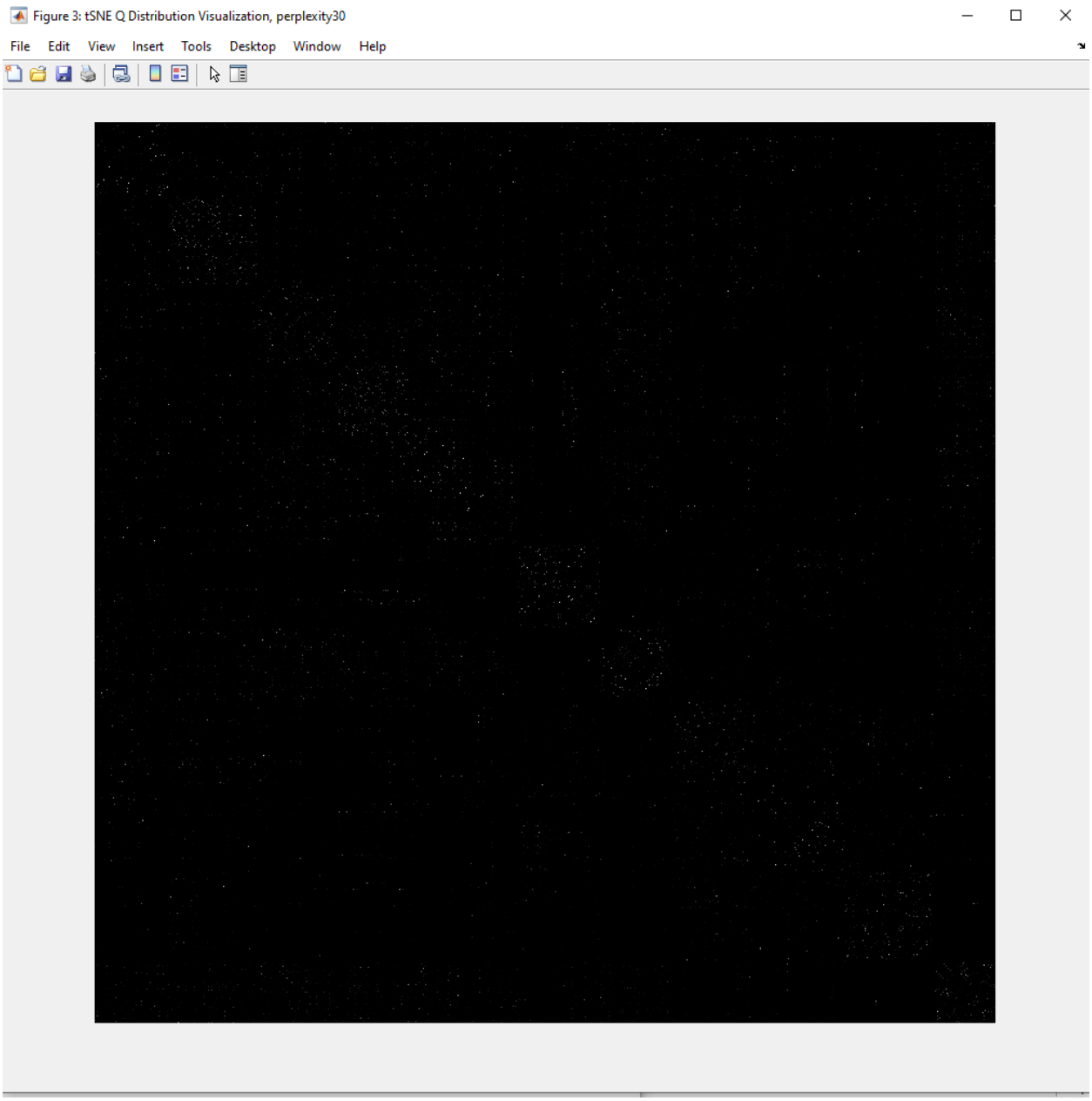


Figure 1: t-SNE operation Perplexity 30
File
Edit
View
Insert
Tools
Desktop
Window
Help

High-dimensional affinity matrix visualization:



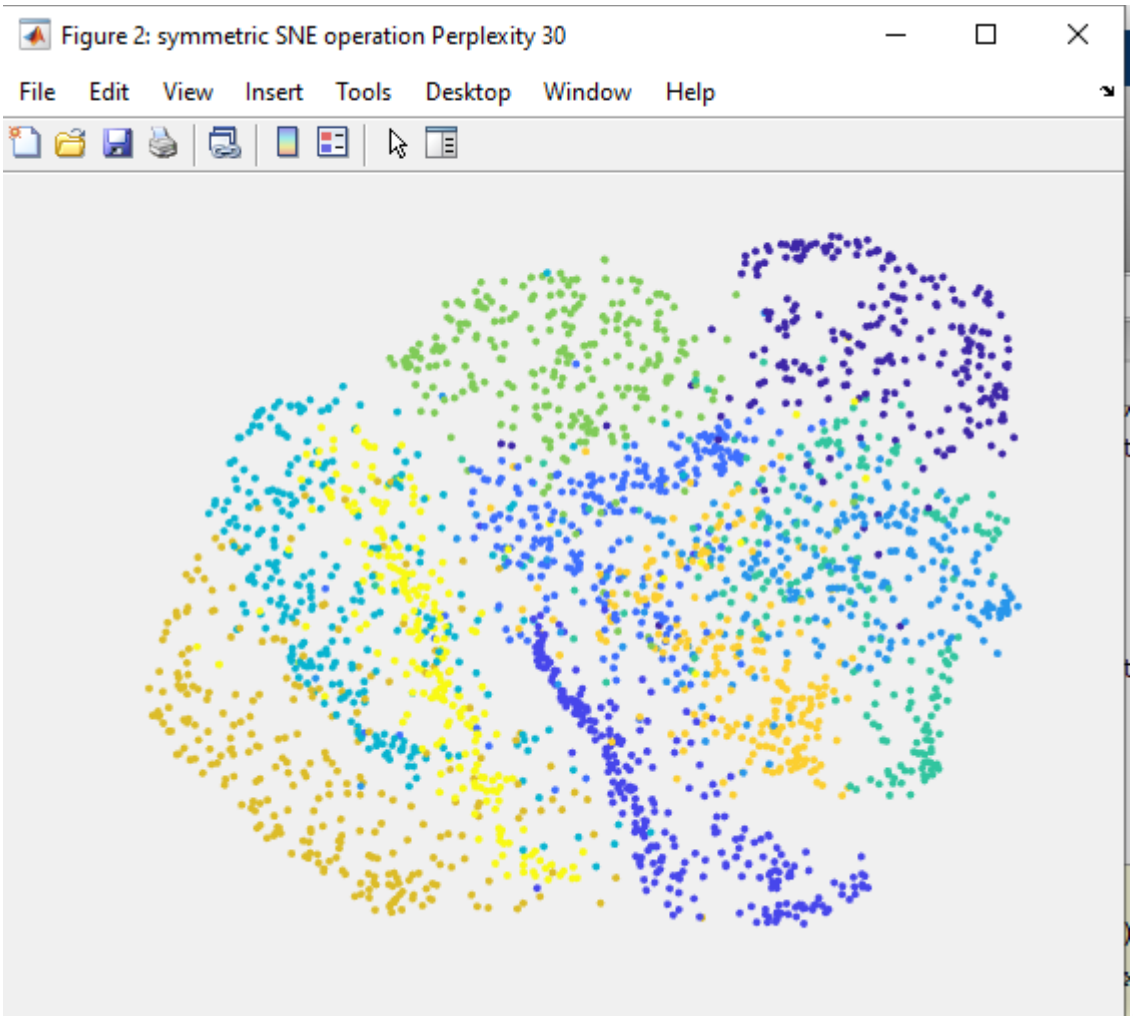
Low-dimensional affinity matrix visualization:



Block structure is starting to be seen in the Q matrix, shows us the number of different classes there

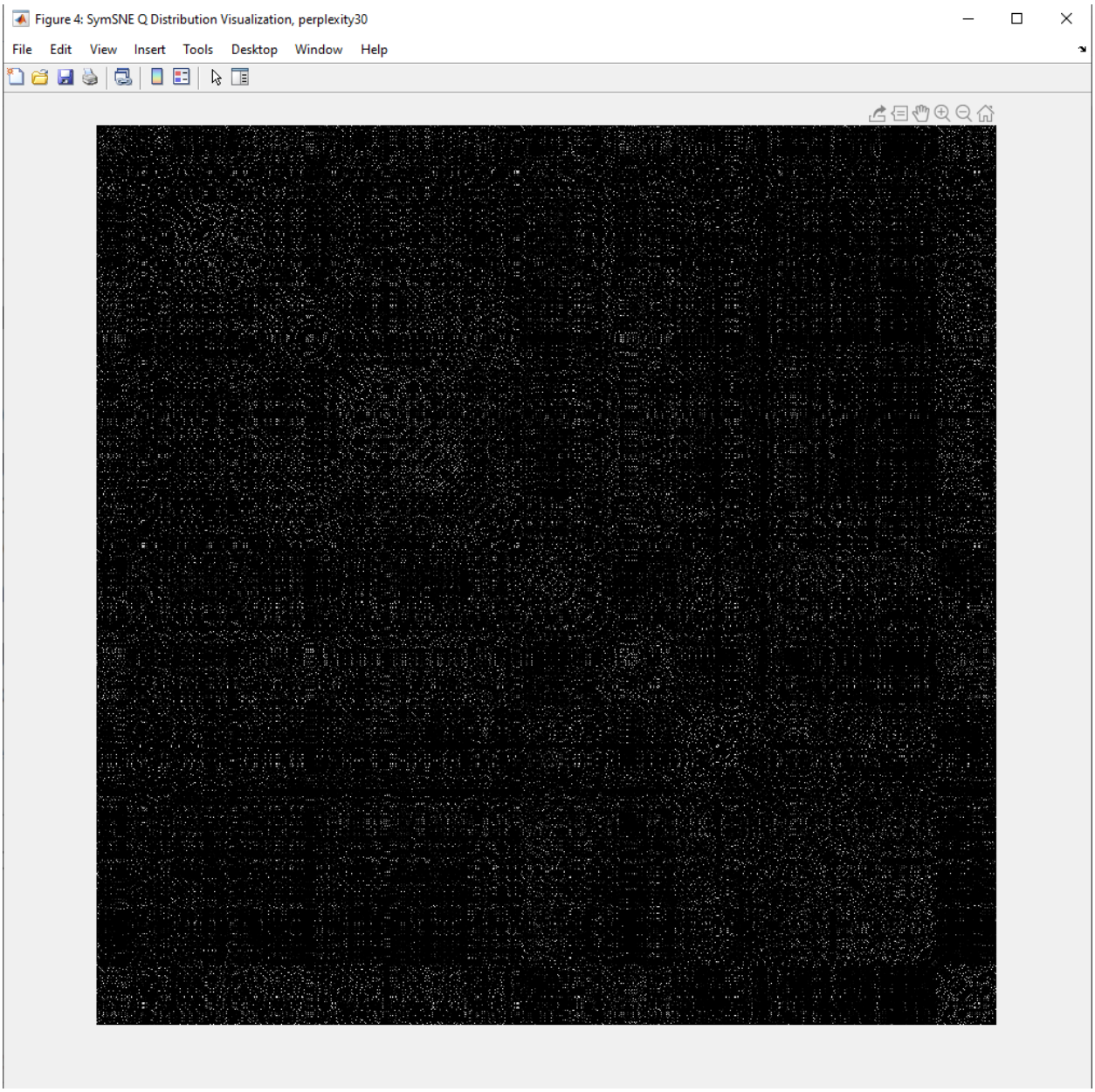
Symmetric SNE:

Final embedding:



We see the difference between t-SNE and symmetric SNE right away: symmetric SNE embedding is far more crowded than t-SNE. This is the problem of **overcrowding**, since for symmetric SNE distance measure points do not need to be too far away to achieve low probability. This is exactly what justifies the use of t-distribution, for which data has to be further away in low-dimension in order to achieve low probability

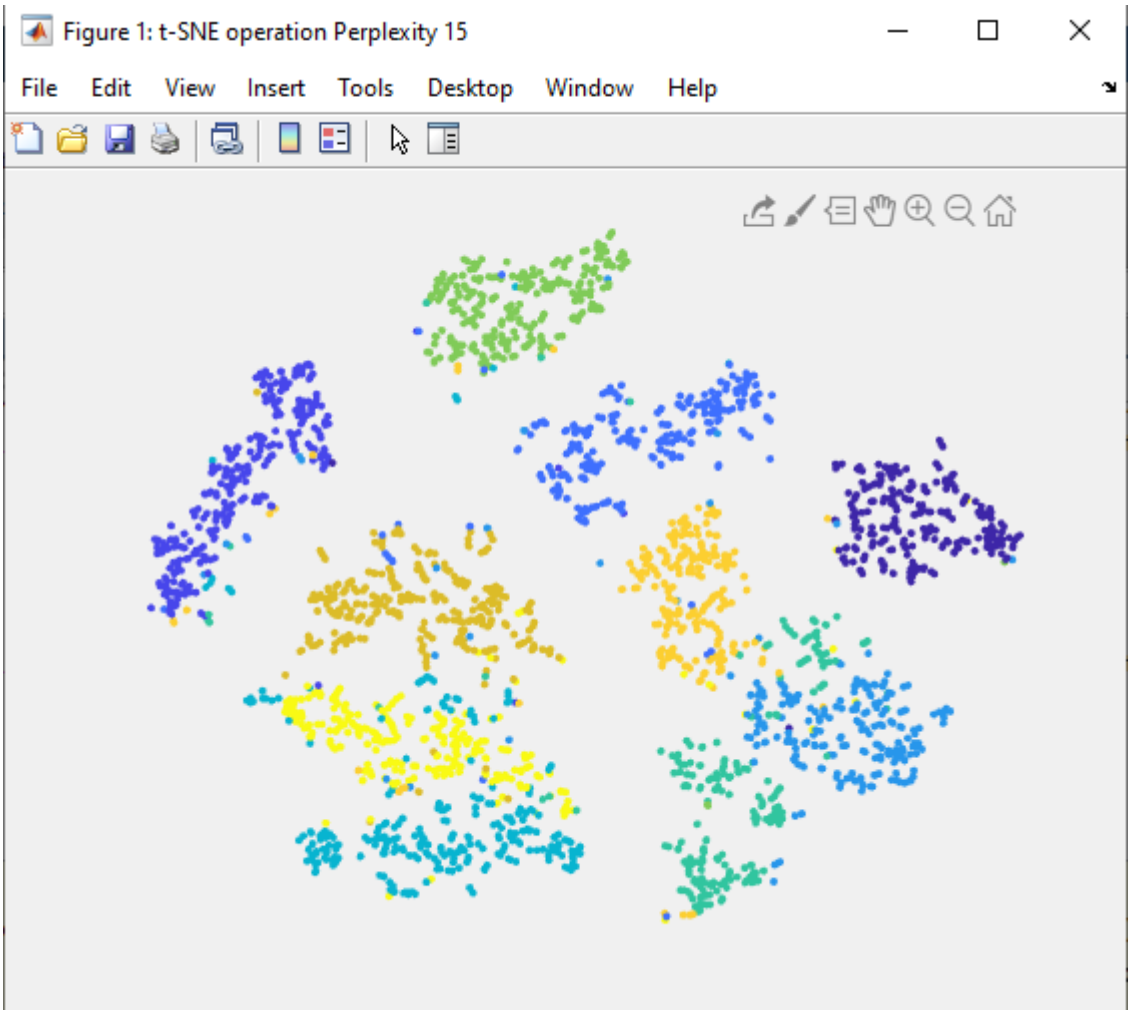
High-dimension affinity matrix remains constant for both approaches, however low-dimensional one is quite different:



We see, that points here have much higher probability, hence the overcrowding problem

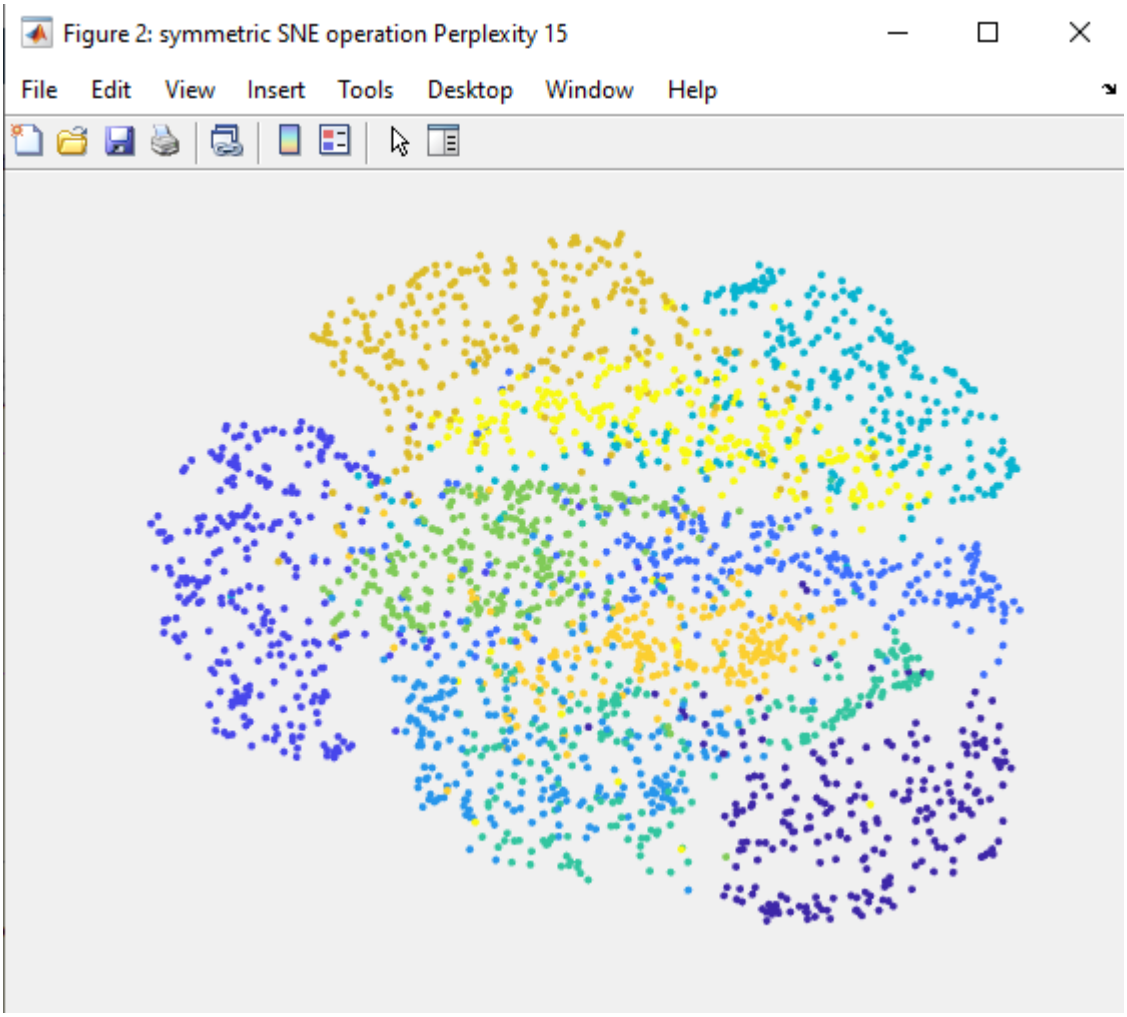
Perplexity 15:

t-SNE:



Scattered more, than with perplexity = 30

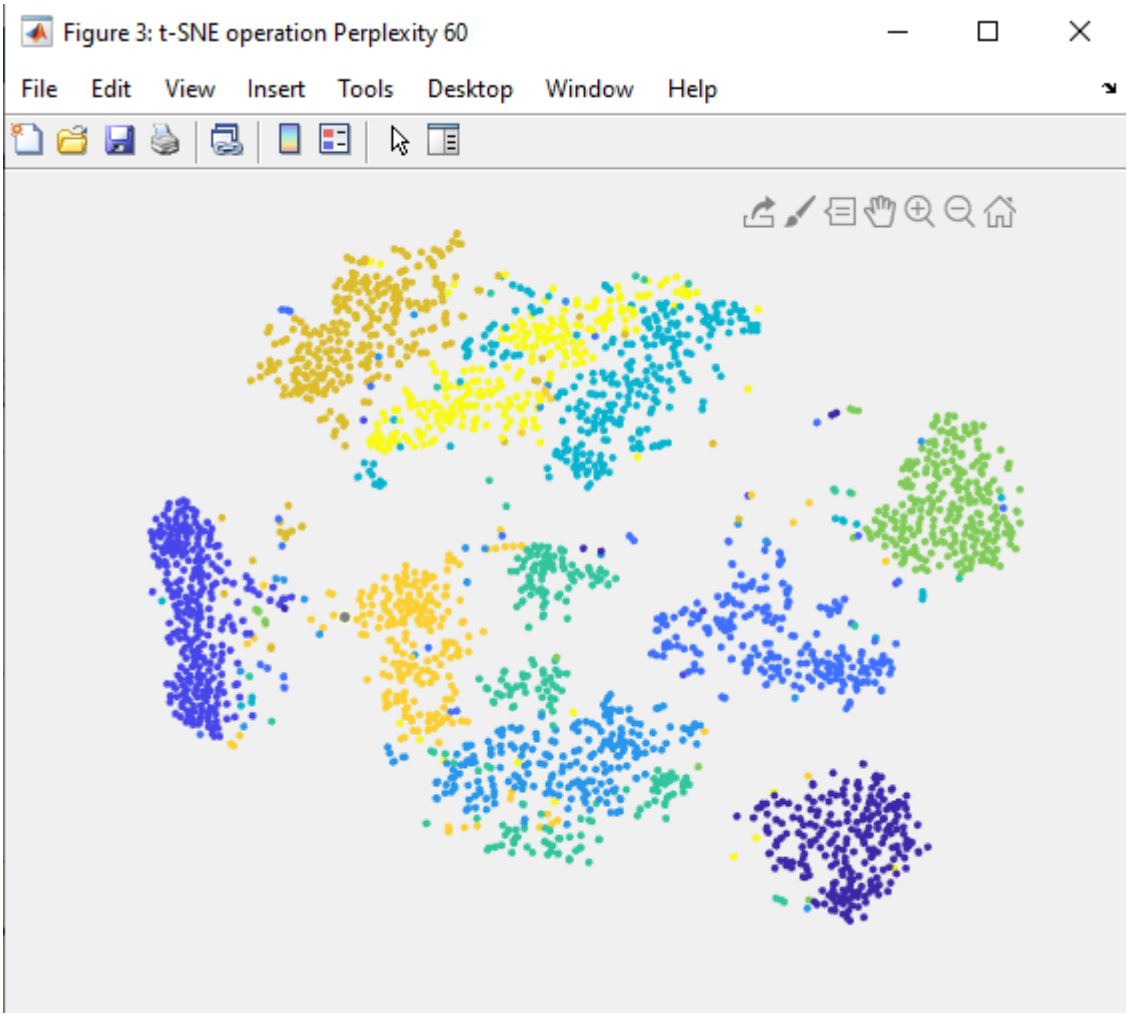
Symmetric SNE:



Similar situation, classes are less grouped together

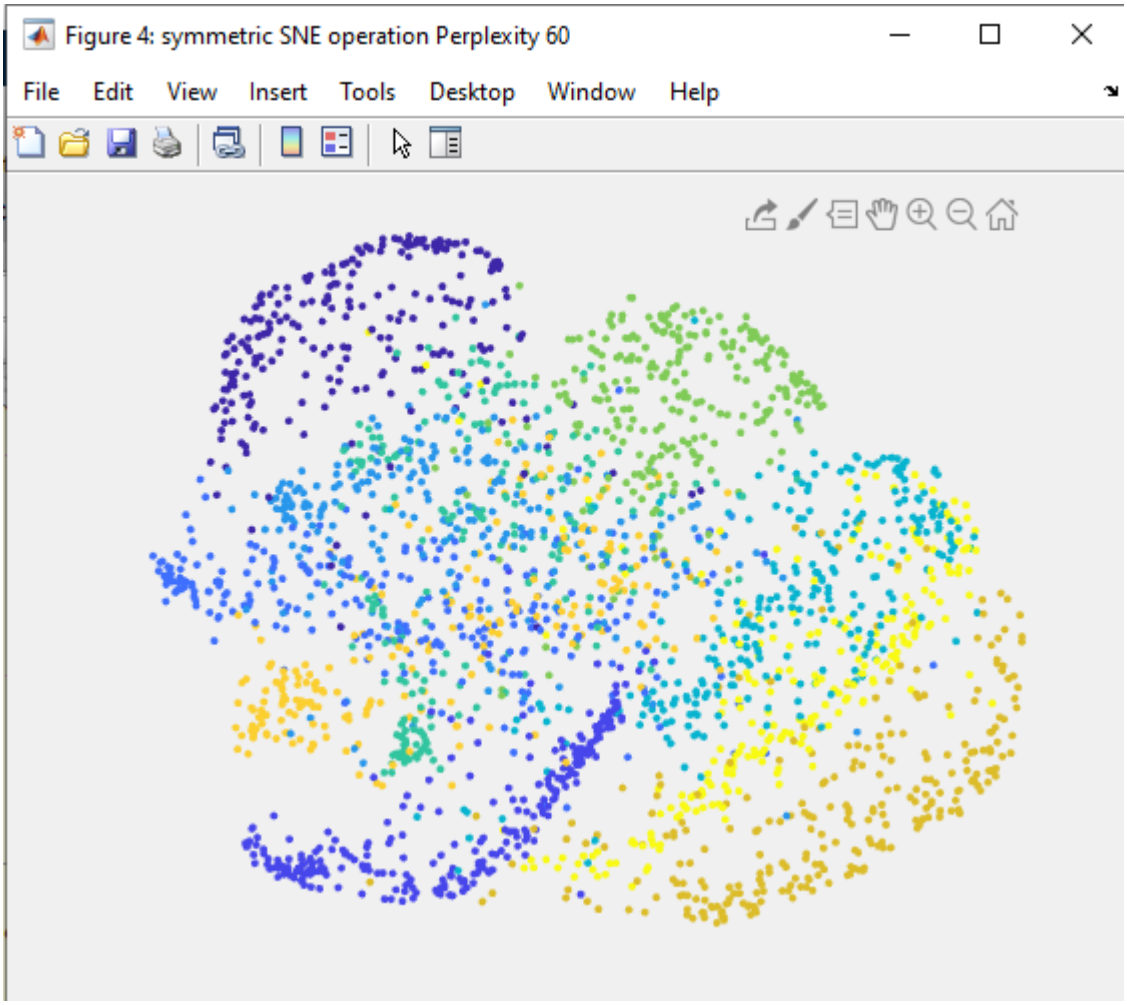
Perplexity 60:

t-SNE:



Increasing perplexity makes our classes more compact in the low-dimensional space

Symmertic SNE:



Classes are grouped, but still crowded