**Machine Learning**

Homework 7

Report

**Name**: Sergiy Shelekh

**ID**: 0760822

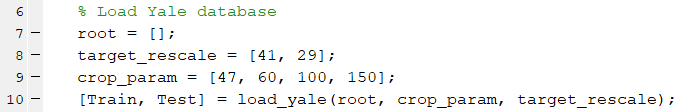
**Email**: [proxitrone@gmail.com](mailto:proxitrone@gmail.com)

**Introduction**

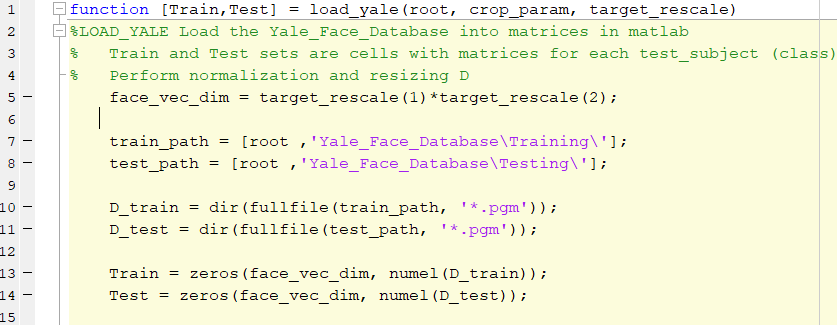
­The first part of this homework is to use Principal Component Analysis (PCA) and Linear Discriminative Analysis (LDA) on the Yale database of facial expression images to find eigenfaces and fisherfaces of those images, be able to classify them. In the second part we are given an implementation of t-SNE 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.

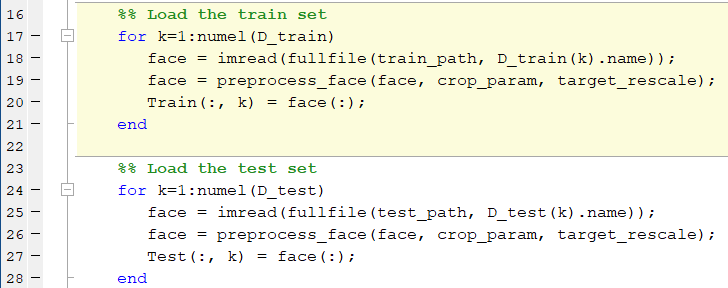
**PCA: Implementation**

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.

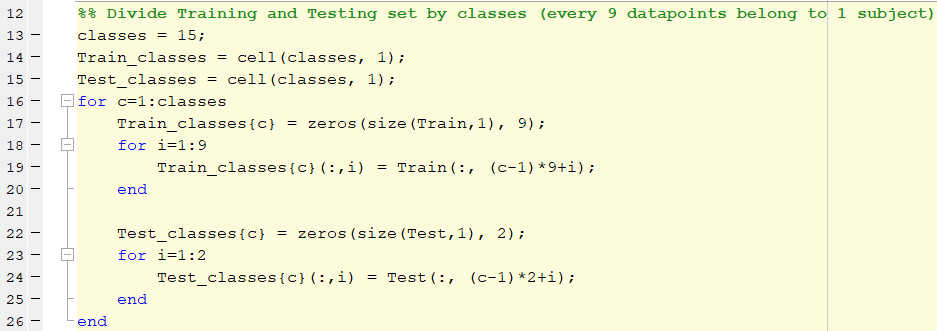


To load all images into matrices right away we take the property of the directory, where they are stored to get the number of images to load. Images are loaded to a single matrix (Train or Test) as column vectors, each column is a separate observation.





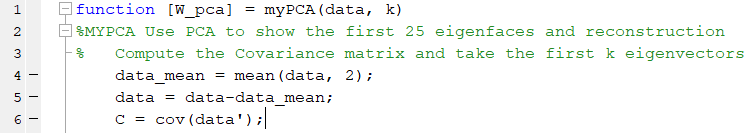
Then, some more processing on data is done, we divide datapoints in both training and testing set into separate classes, this will come in handy during LDA and for classification.



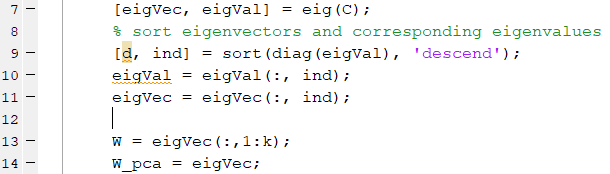
After all preprocessing is done, we can go to the PCA part.



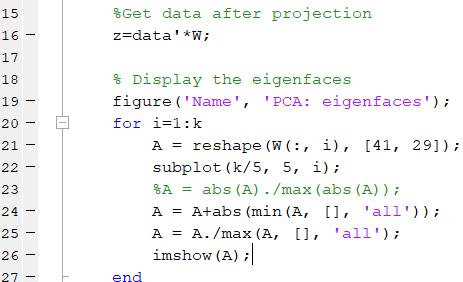
PCA procedure is quite straightforward, we first zero-mean our data, and then compute it’s covariance:



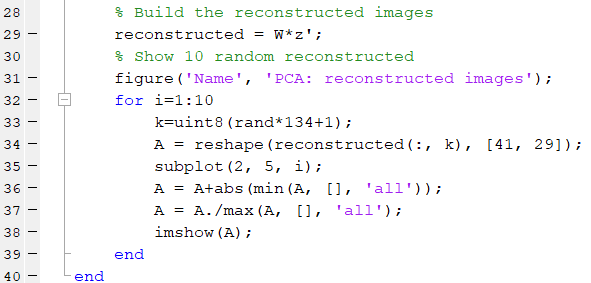
This covariance matrix or total scatter plays significant role in PCA, since a number of first largest eigenvectors of this covariance gives us exactly the orthogonal projection weights we need.



We then show the resulting eigenfaces

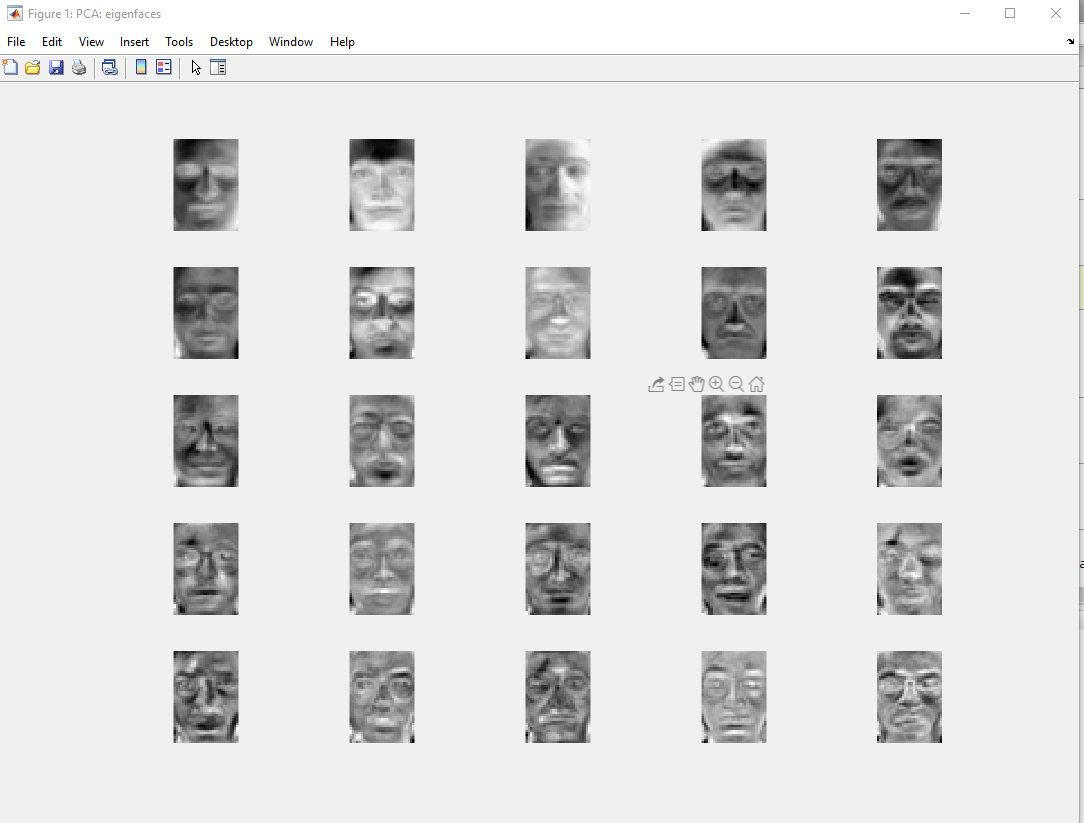


And 10 random reconstructed images:

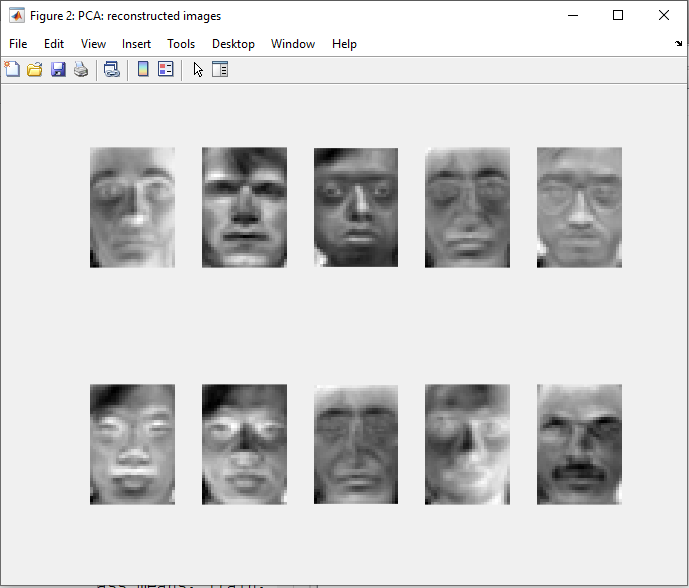


**PCA: Results**

We show 25 eigenfaces:



And 10 random reconstructed images:



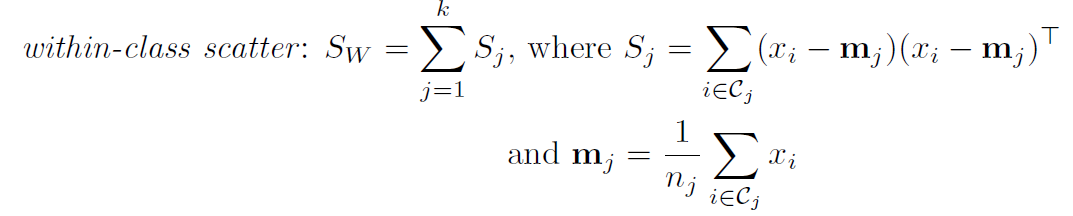
This approach gives us some nice results, eliminating varying lighting conditions present in the original dataset

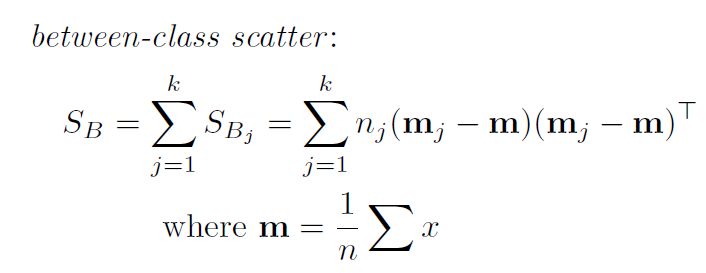
**LDA: Implementation**

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

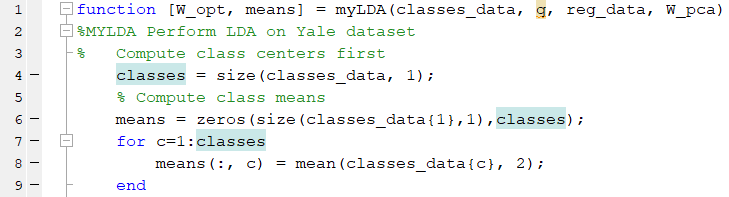


LDA is a kind of supervised algorithm, where we want to project our high-dimensional data into some low dimension, while maximizing between-class scatter and minimizing the within-class scatter. Both these values can be easily computed:

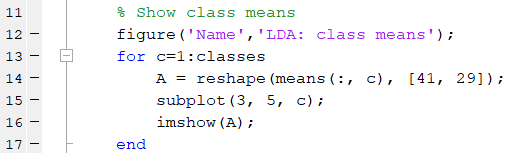




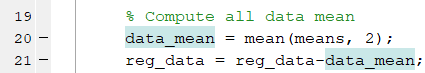
And that’s exactly what we are doing next



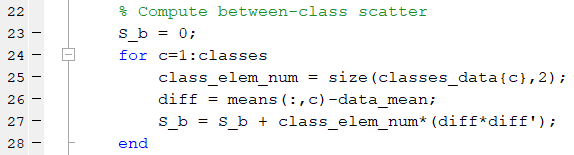
First, compute class means and them display them for our reference

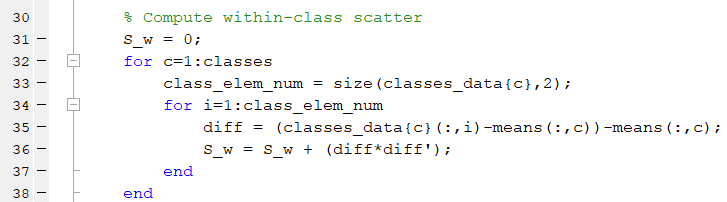


We then proceed to computing all data mean and use it to zero-mean our data

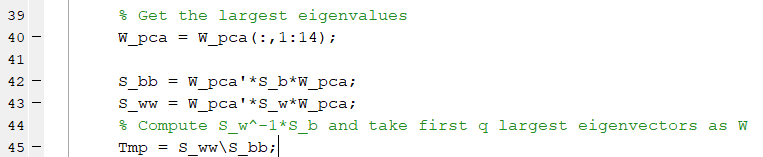


Now we can start getting those within- and between-class scatter matrices:

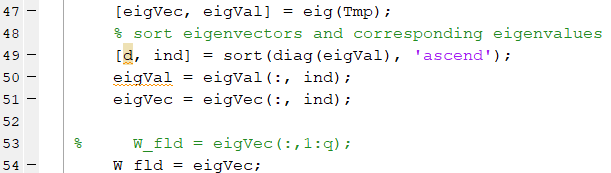




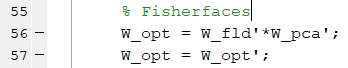
Originally, we would simply take first largest eigenvalues of  as our projection weight matrix, but in our case is singular, which makes the inverse undesirable. So, we use the result from PCA to overcome that singularity problem. Project both within- and between-class scatters by the PCA projection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



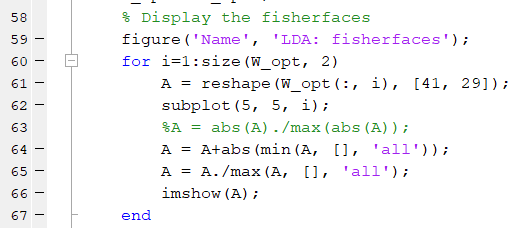
Take the largest eigenvalues as the Fisher’s linear decomposition weights:



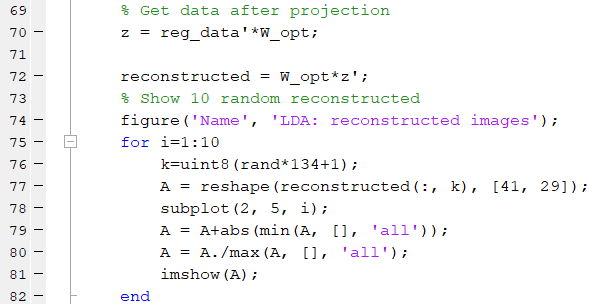
And then combine this projection with PCA to get optimal weights and our fisherfaces:



Show them first

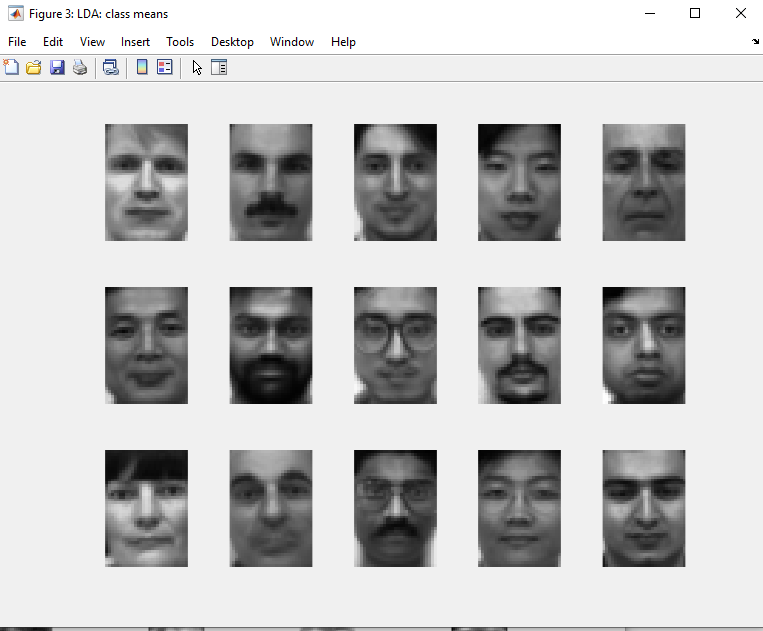


And then perform reconstruction:

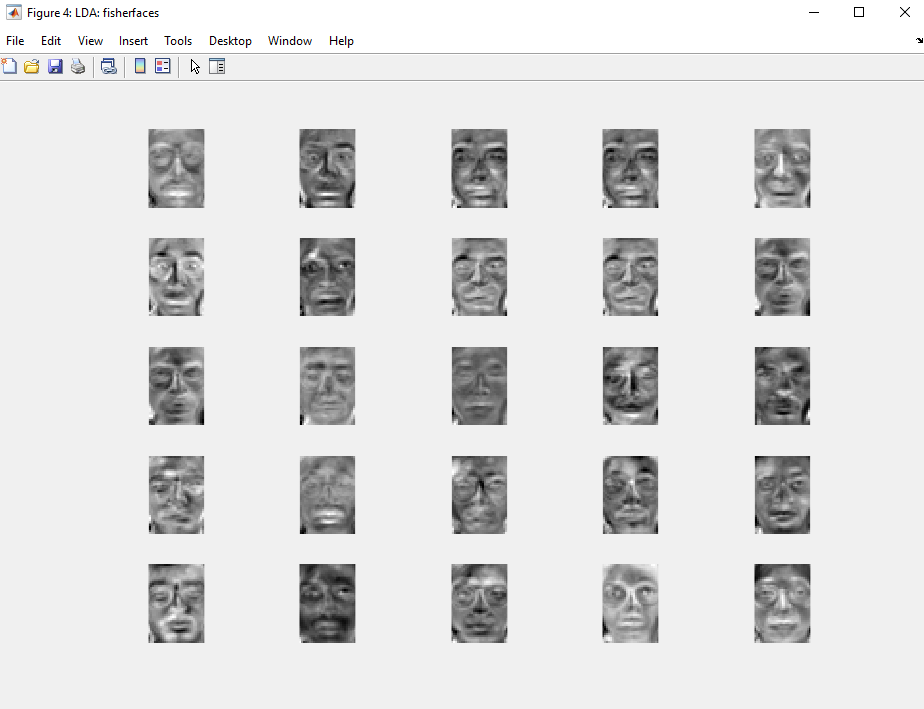


**LDA: Results**

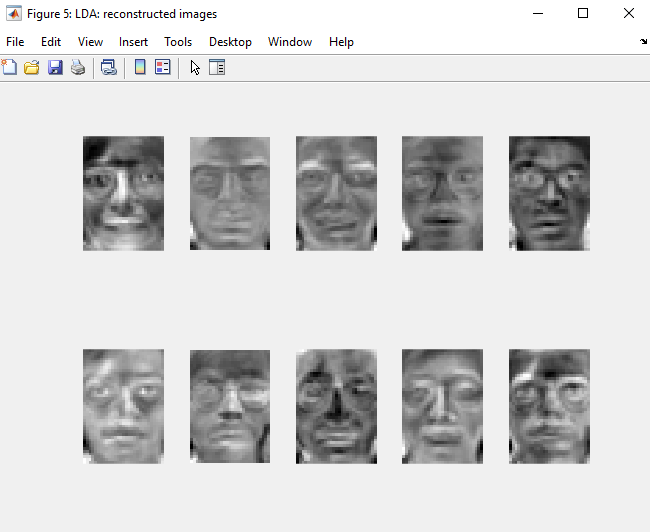
Class means:



Fisherfaces:

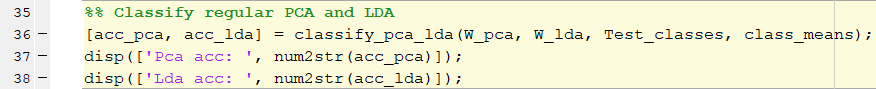


Reconstructed images:

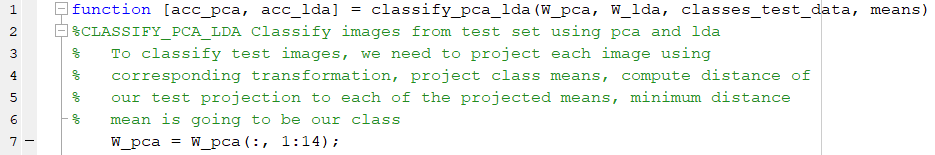


**PCA&LDA Classification: Implementation**

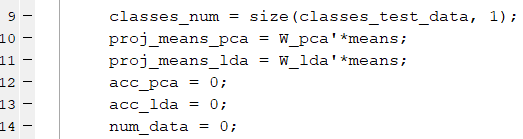
We now have both projection matrices from PCA and LDA, so we can go on to classifying images from our testing set



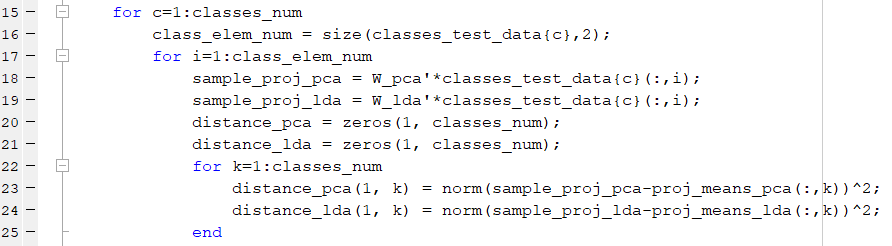
In this classification we project our testing datapoint onto the low-dimensional spaces, given by PCA and LDA, and compute the distance between this projection and projected class means.



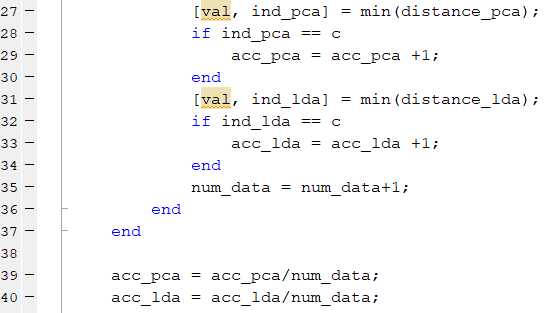
Choose first largest components, project class means:



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.



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



**PCA&LDA Classification: Results**

­Resulting accuracies are quite high:



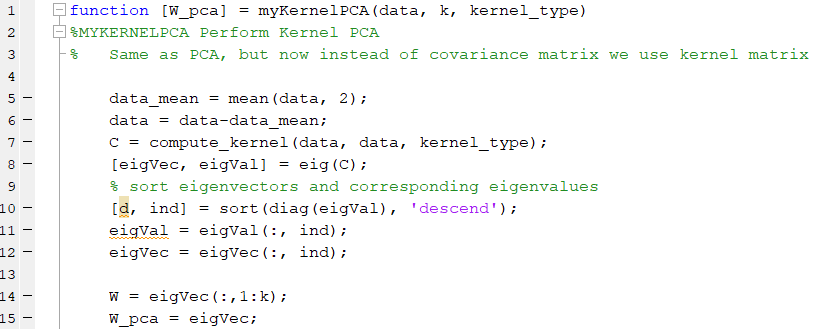
LDA performs better than PCA, which is expected, since it’s a more accurate supervised technique

**Kernel PCA&LDA: Implementation**

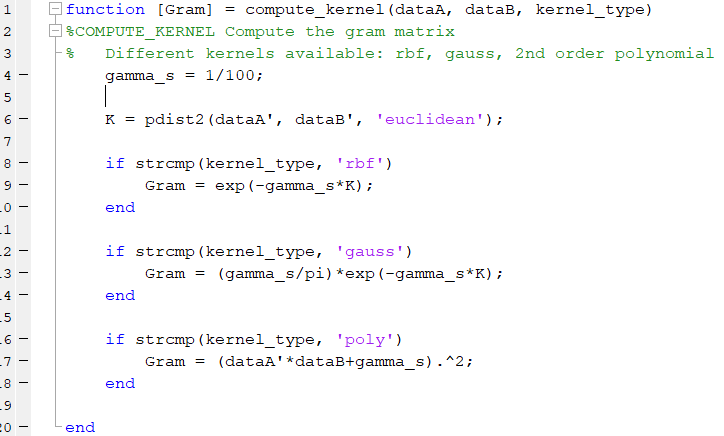
After we have finished with the regular PCA and LDA, where we only use linear distances as our datapoint correlation, we move on to a more elaborate version of the above two algorithms, where we, once again, use the kernel trick to allow the boundaries to be nonlinear. I use two different kernels to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



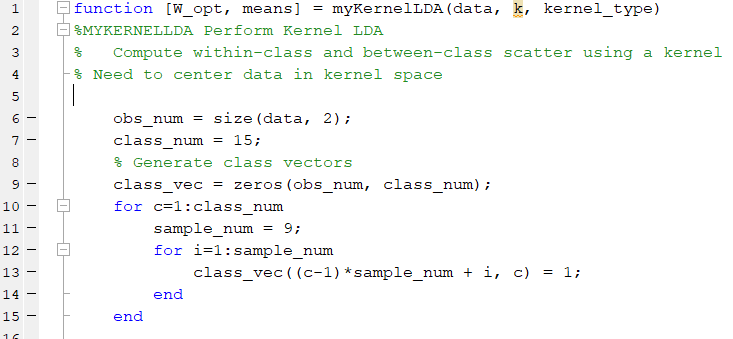
Kernel PCA is done quite similar to regular PCA, but instead of the regular covariance matrix, we use pairwise distances between points in kernel space



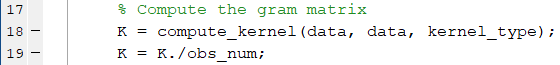
We compute all kernels in a similar fashion as in previous assignments



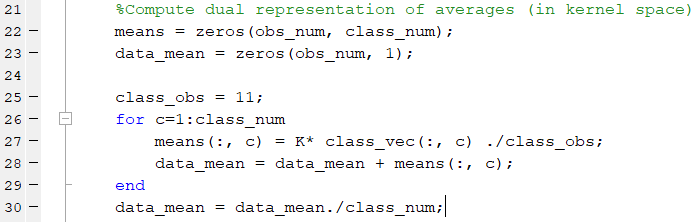
Implementation of kernel LDA is a bit more interesting. We now work in the feature space, where we also need to find means of our classes and distances between points. To simplify computations, we take advantage of matrix operations and we need a class matrix, where rows are observations and columns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



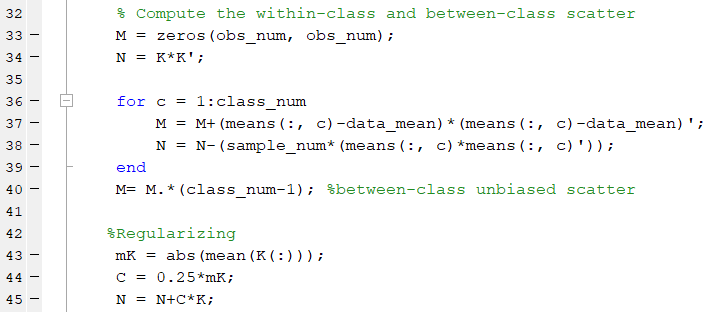
We then compute the Gram matrix and regularize it



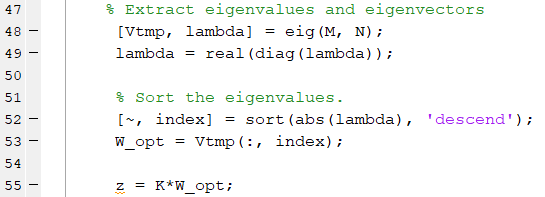
Find the means of our classes in feature space



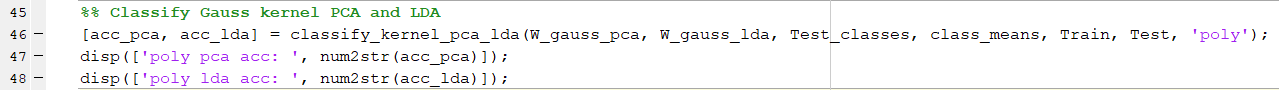
Now we have everything we need to compute within- and between-class scatter in the feature space. Notice the extensive use of our Gram matrix, no additional computations



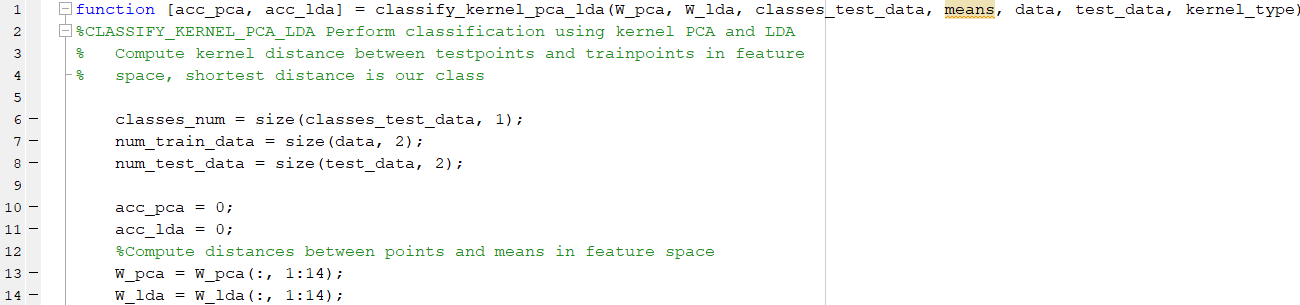
After regularizing the within-class scatter, we solve the generalized eigenvector problem with the two different scatters we obtained:

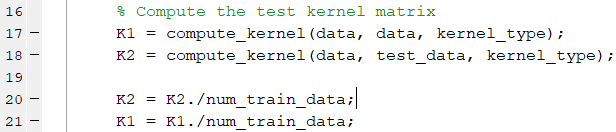


And, as before, our projection weights are given by the eigenvectors and we can go on to classification.

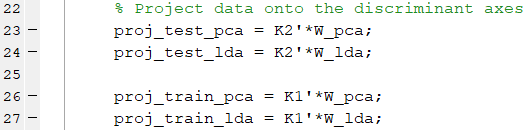


Since we are now working in the feature space, we can’t simply project class means onto the low dimension, we need to compute the kernel between testing and training points, project those points onto a low-d space, and compare distances there.

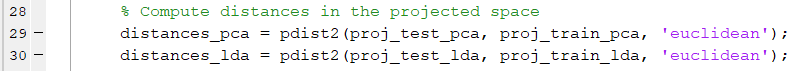




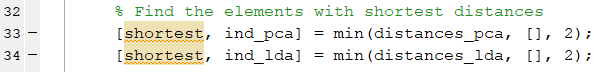
Project data onto the discriminant axes:



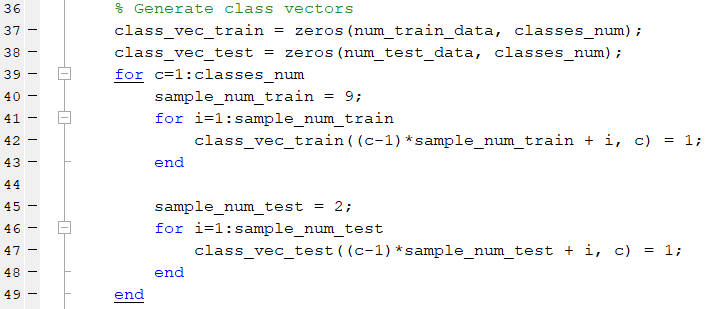
Compute distances in the projected space:



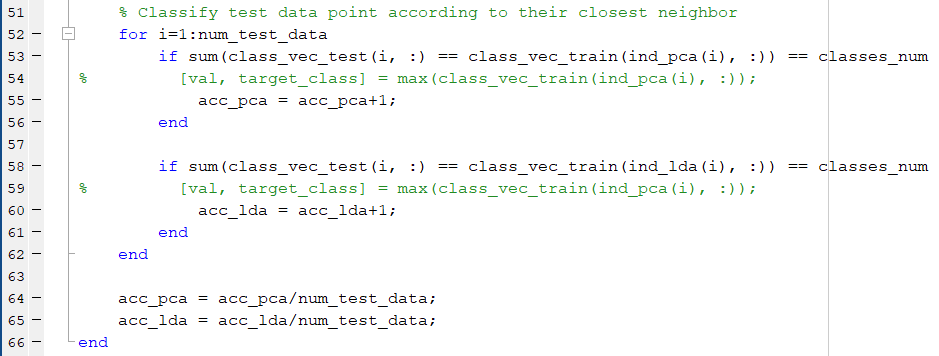
Find which distances are the shortest



Generate class vectors to compare classifications:

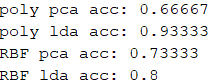


And finally classify the points and update accuracies:



**Kernel PCA&LDA: Results**

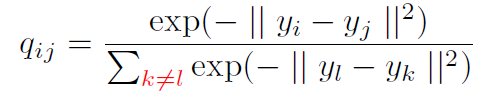
The results are as following:



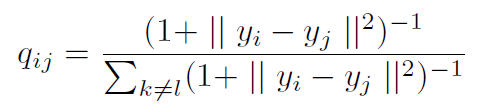
We see, that for polynomial kernel the accuracy decreased for PCA compared to regular version, which is most likely due to the inaccurate choice of hyperparameters. LDA shows same results in case of polynomial kernel, but lower accuracy for RBF. PCA rbf kernel performs better than polynomial

**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:

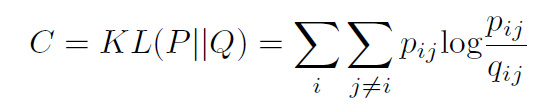


While in t-SNE we modify that distribution to be of t-distribution, which allows points with low probability to be farther away from each other in that low-dimensional space:

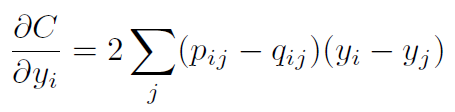


The next difference is the gradient of the objective function we want to minimize (KL divergence).

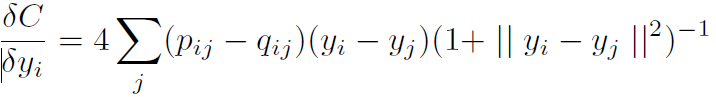
Objective:



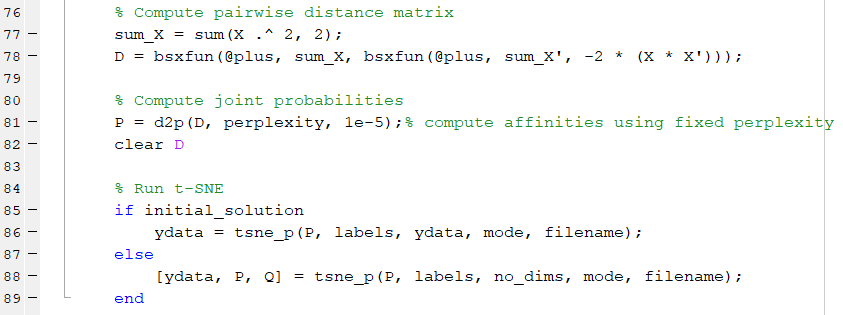
In case of symmetric SNE, the gradient is given by:



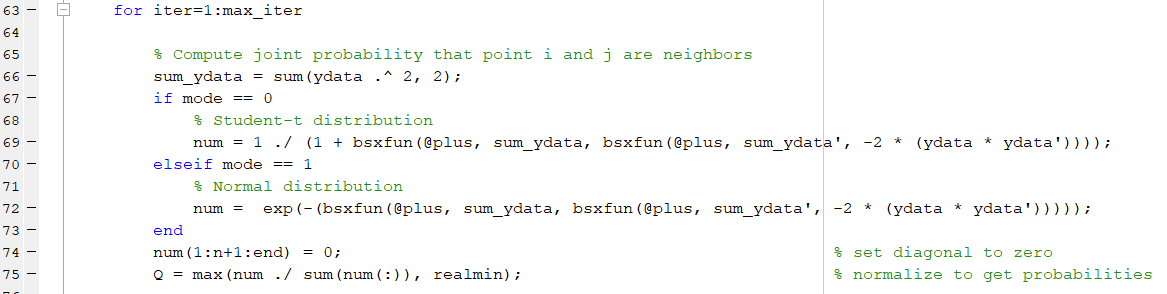
While for t-SNE we have:



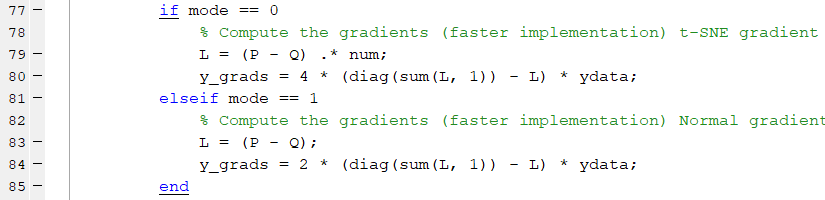
Now we go on to identify and change the two corresponding parts in the given source code. I won’t go into the detail of implementation of the given code, but generally, the program first normalizes the input and computes pairwise distances between points, to be further used in computing joint probabilities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



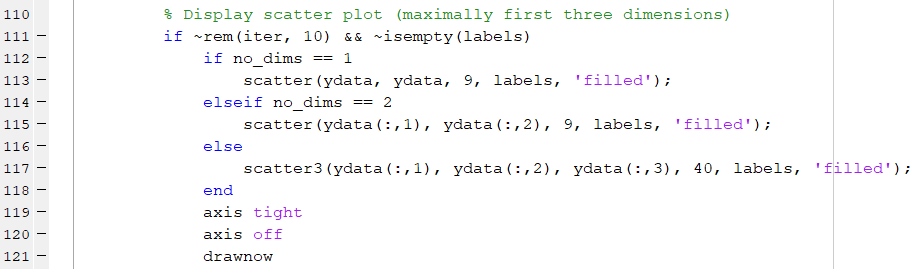
After we have the constant affinity matrix P, we go on to compute the low-dimensional affinity matrix Q, gradient of KL divergence w.r.t Q, and do a gradient step to eventually converge to a local optimal solution

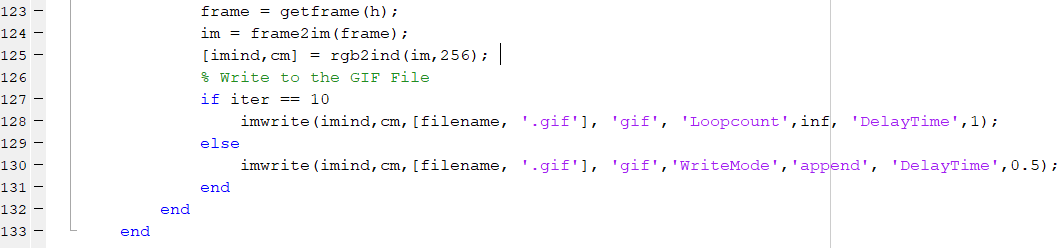


Here we can see the computation of matrix Q for different distributions, student t-distribution is given by the authors, and normal distribution is our modification. Computation is pretty straightforward

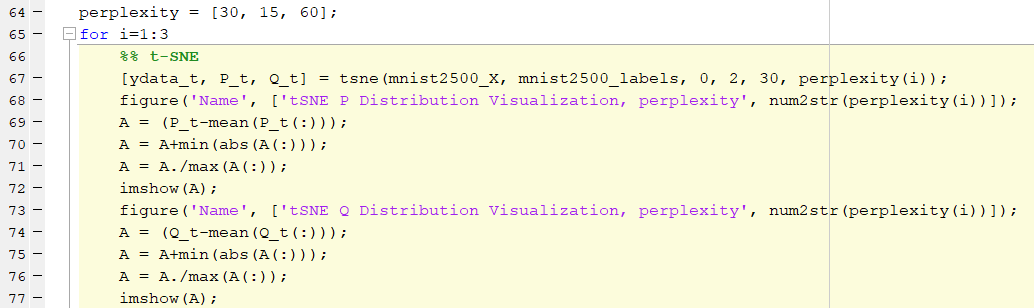


This part is responsible for computing the gradient of KL divergence w.r.t matrix Q. Our modification is in computing the Normal gradient. Later in the program, the solution (ydata) is updated using the above gradient. The progress is visualized every 10 iterations and we also modify the visualization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

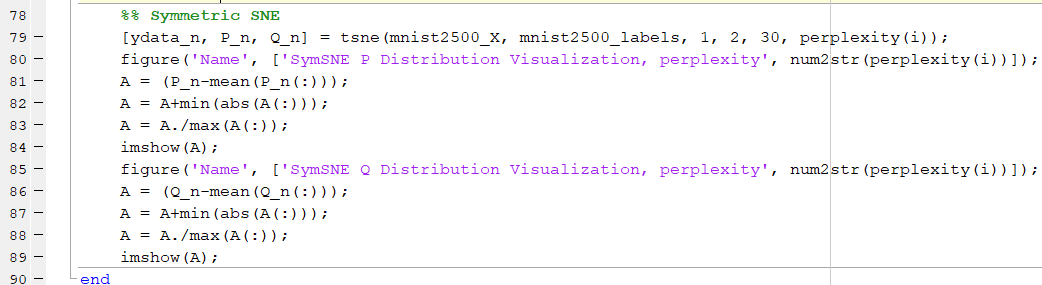




We do the optimization for a total number of 1000 iterations and then output distribution pairwise similarities in both high- and low-dimensional spaces to visualize the result in the end.



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



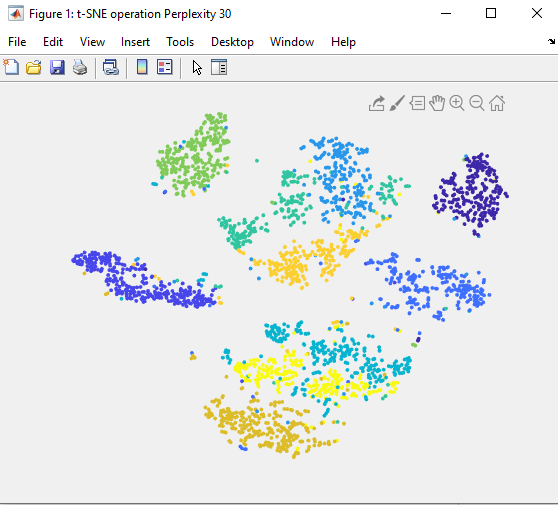
**t-SNE and Symmetric SNE: Results**

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

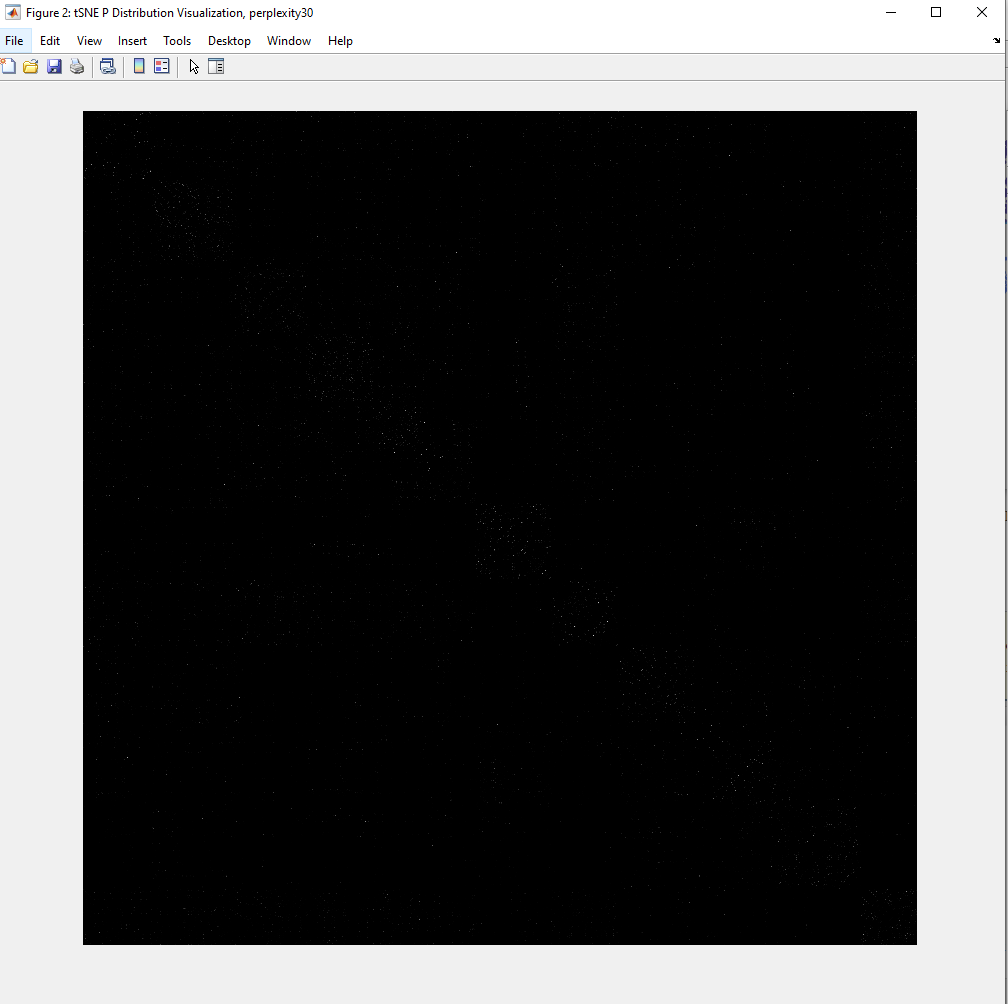
Perplexity 30:

t-SNE:

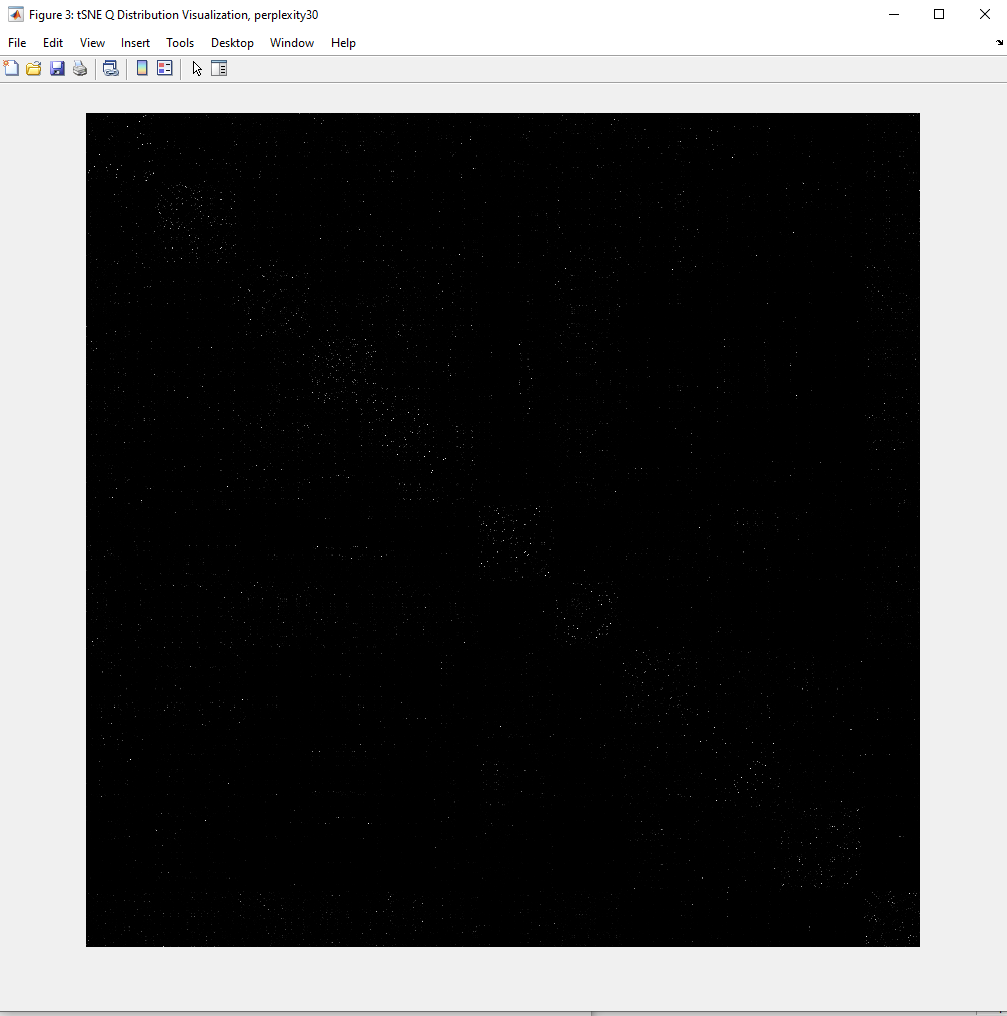
Final embedding



High-dimensional affinity matrix visualization:



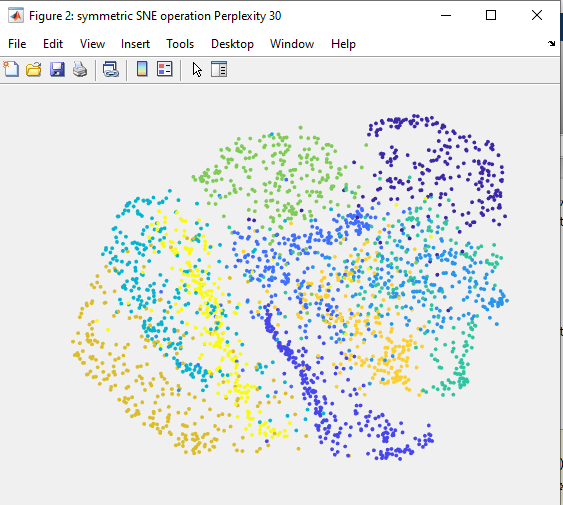
Low-dimensional affinity matrix visualization:



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

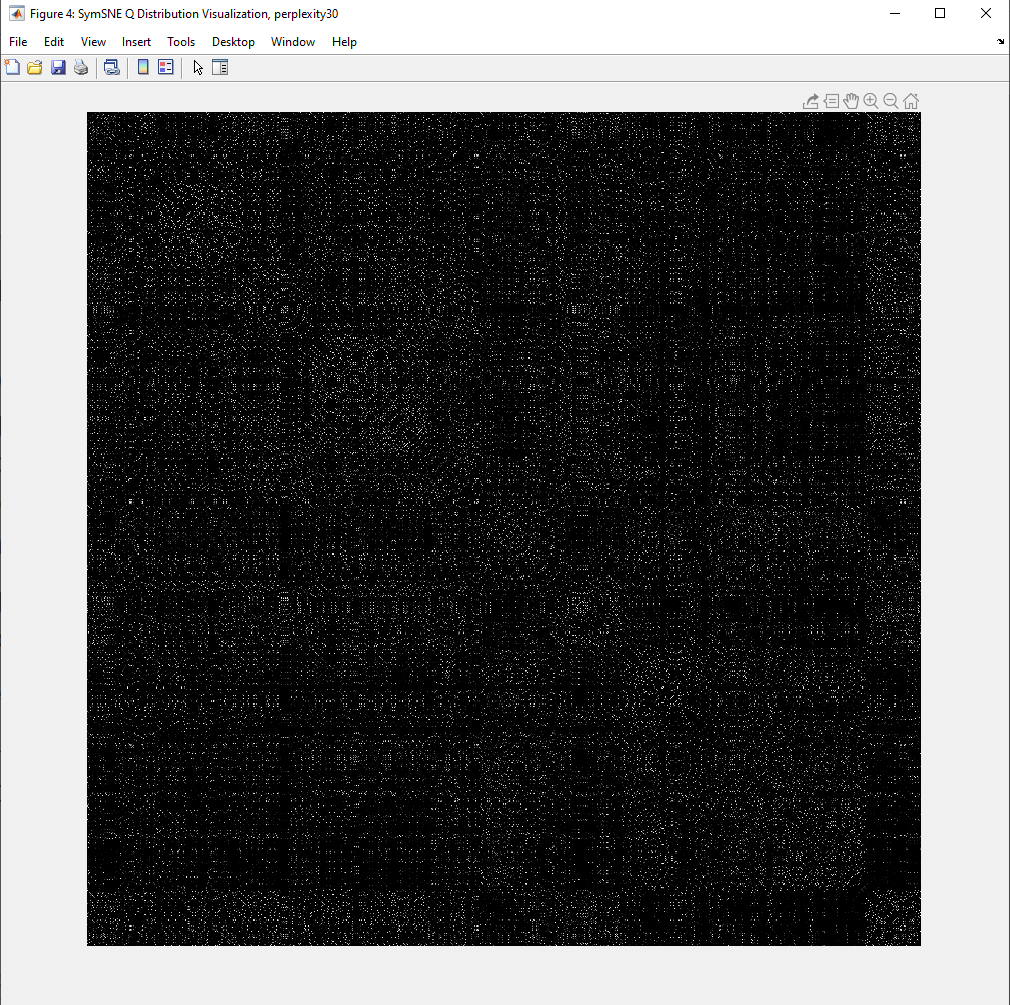
Symmetric SNE:

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

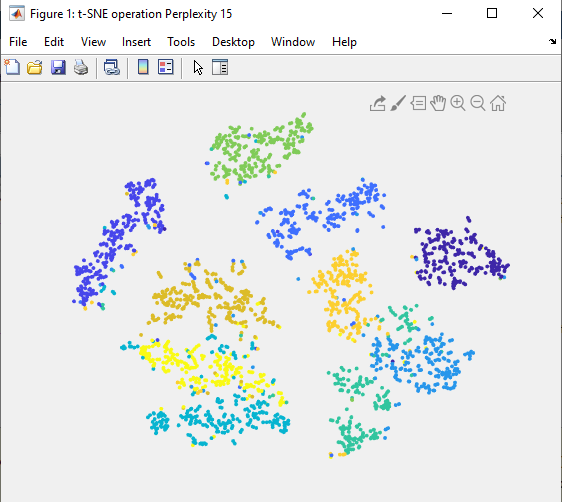
High-dimension affinity matrix remains constant for both approaches, however low-dimensional one is quite different:



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

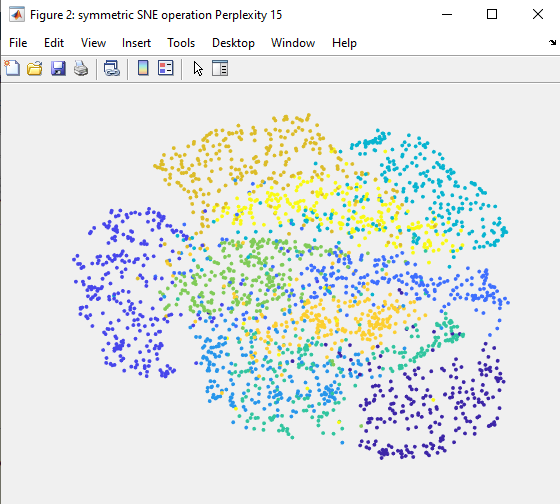
Perplexity 15:

t-SNE:



Scattered more, than with perplexity = 30

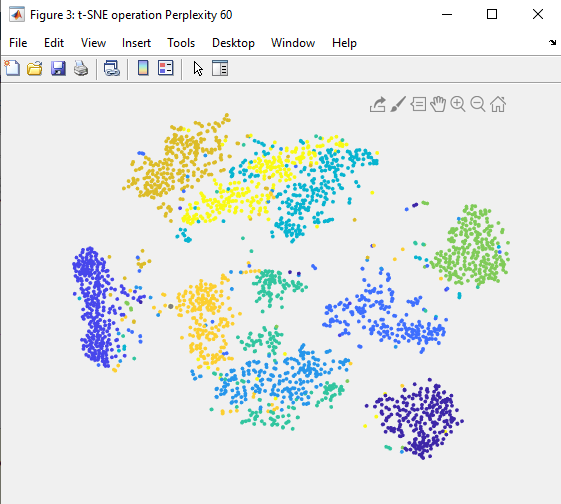
Symmetric SNE:



Similar situation, classes are less grouped together

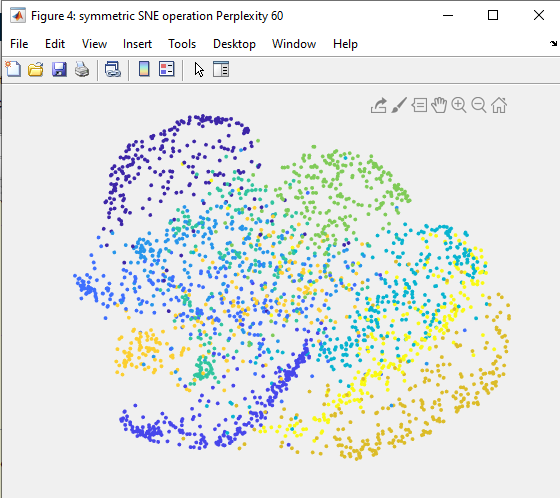
Perplexity 60:

t-SNE:



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

Symmertic SNE:



Classes are grouped, but still crowded