Introduction to Intelligent Systems
Lab Session 6

Group 30
Sergio Calogero Catalano (s3540294) & Emmanouil Gionanidis (s3542068)

October 23, 2017

Introduction

Supervised learning aims to minimize the generalization error, which is the error with respect to new data. To evaluate how good a model can generalize, we must use some kind of validation process. One of the best validation processes is the n-fold validation, and it is what we implemented for this assignment. We used the same code from the last assignment, with small changes needed to adapt to the new requirements (this time the function simulating the epochs could not create the plots itself, because it needed also the test data and some other pieces of information).

Remarks

We noticed that the difference between the learning curves of the test errors, for different folds, was really large, and, even if we were expecting some differences, there were some folds with error rates of 0.05 and some others with error rates of 0.5 during the same execution (more substantial differences than what we expected). Thus we plotted some figures with the example data and the prototypes, and we realized that the low error rates occurred especially when the test points were trivial (near the mean of the points of their class) and that the large error rates were due to test points near the mean of the points of the opposite class. This helped us understand that the choice of the test sets has a greater influence than what we imagined, and that it is thus important to consider the means of these values, and none of them individually.
In the following sections we refer to the error rate simply as error, because we never use the actual error values, but only the error rates, and we thought that, specifying that they were rates everytime we mentioned them, was not necessary.

The algorithm implementation

The function in **Listing 1** implements the n-fold validation, using the following input parameters:

- data: the given data set, with an added column indicating the class of each example point
- n: the number of folds
- h: learning rate
- P: number of examples in the data set
- K: number of prototypes
- t_max: number of epochs

Listing 1: *Function that executes the n-fold validation*

```
function n_fold_validation(data, n, h, P, K, t_max)
```

At the beginning of the function, we shuffle the data rows and we divide them into n subsets, as shown in **Listing 2**

Listing 2: *Code to shuffle the data and to create subsets*

```
% Randomly shuffle the data
data = data(randperm(P),:);

% Divide the data in n different subsets
subsets_size = P/n;
subsets = zeros(subsets_size, 3, n);
```

```
    for i = 1:n
        subsets(:, :, i) = data(subsets_size*(i-1)+1:subsets_size*i,:);

    end
```

Then we initialize the required vectors and matrices and execute the required operations, as shown in **Listing 3**

Listing 3: *Code that executes the learning procedure and calculates the test and training errors*

```
    % Initialize training and test errors
    training_errors = zeros(1, t_max);
    test_errors = zeros(1, t_max);
    training_errors_std = zeros(1, t_max);
    test_errors_std = zeros(1, t_max);
    training_set = zeros(subsets_size * (n-1),3);

    %Execute for each fold
    for i = 1:n
        %Create test set and training set
        test_set = subsets(:,:,i);
        tmp_set = subsets(:,:,[1:i-1 i+1:n]);
        for k = 1:n-1
            training_set(subsets_size*(k-1)+1:subsets_size*k,:) =
tmp_set(1:subsets_size, :, k);
        end

        % Execute learning algorithm and return training and test
error
        [prototypes, training_error, test_error] =
completeLearningProcess(training_set, test_set, h, K, size(
training_set,1), t_max, 0, 0, 0);

        %Update sum of errors and sum of squares, which will be used
to compute means and standard deviations later
        training_errors = training_errors + training_error;
        test_errors = test_errors + test_error;
        training_errors_std = training_errors_std + training_error.^2 ;

        test_errors_std = test_errors_std + test_error.^2 ;
    end

    %Compute means and standard deviations and return the necessary
values
    training_errors =  training_errors/n;
    test_errors =  test_errors/n;
    training_errors_std = sqrt(training_errors_std/n - (
training_errors).^2);
    test_errors_std = sqrt(test_errors_std/n - (test_errors/n).^2);
    final_training_error = training_errors(t_max);
    final_test_error = test_errors(t_max);
```

The code is self explainatory thanks to the comments. The only notable thing is that the means and the variances have been computed manually, instead of using the given functions mean() and

std(), so that we did not have to save all the measured values of training and test errors, but could just keep track of their sum.

Overfitting and underfitting

We executed the algorithm with 2, 4, 6 and 8 prototypes and obtained the plots in **Figure 1** , **Figure 2** , **Figure 3** and **Figure 4** , respectively.

Learning curves of mean training and test errors with 2 prototypes

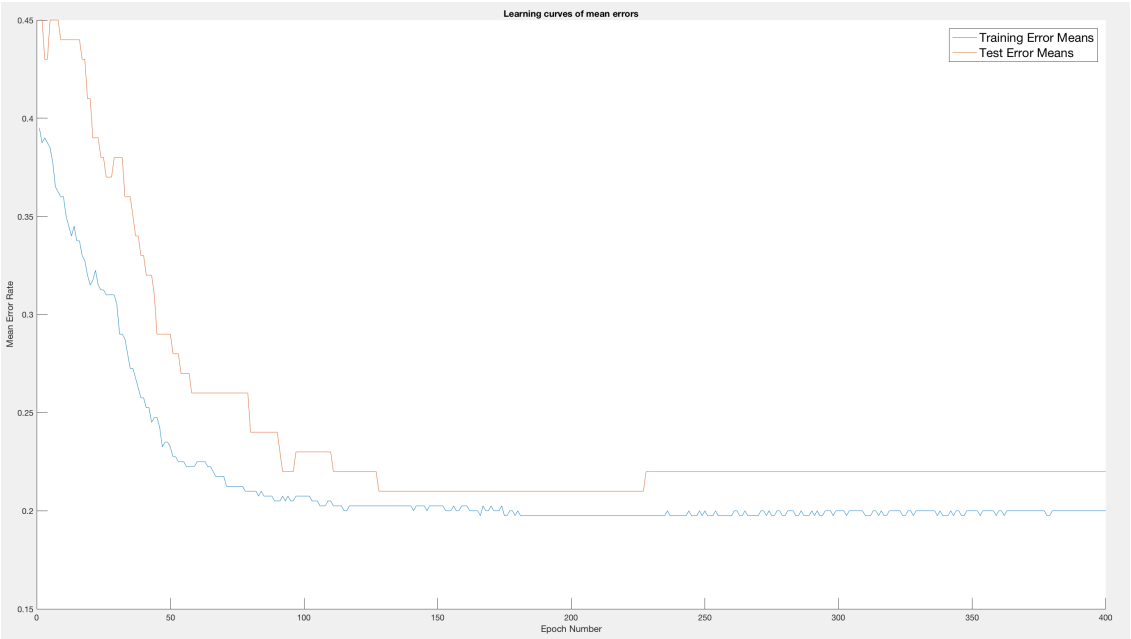

Figure 1: *shows the epoch number vs the mean training and test error rates when the algorithm is executed with 2 prototypes and a learning rate of 0.001*

Learning curves of mean training and test errors with 4 prototypes

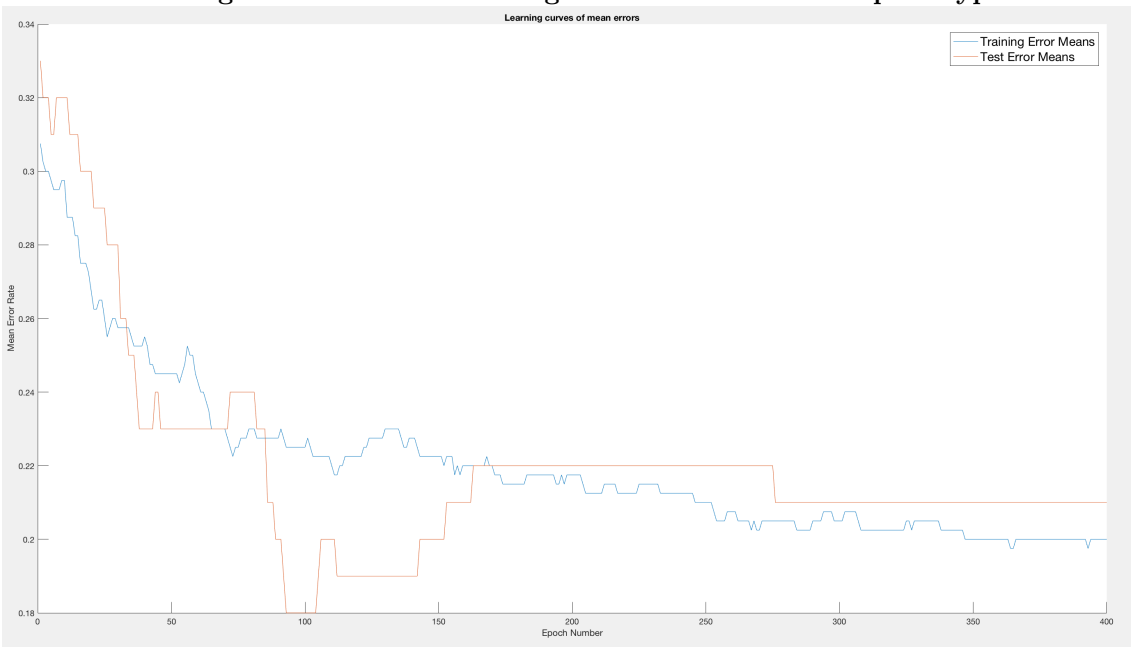

Figure 2: *shows the epoch number vs the mean training and test error rates when the algorithm is executed with 4 prototypes and a learning rate of 0.001*

Learning curves of mean training and test errors with 6 prototypes

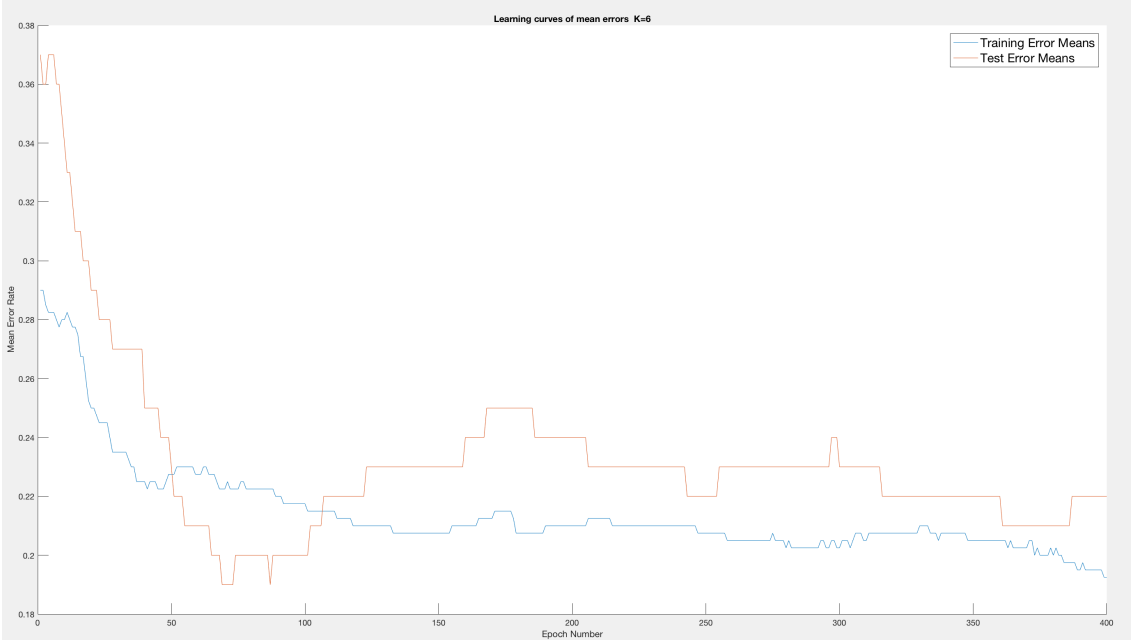

Figure 3: *shows the epoch number vs the mean training and test error rates when the algorithm is executed with 6 prototypes and a learning rate of 0.001*

Learning curves of mean training and test errors with 8 prototypes

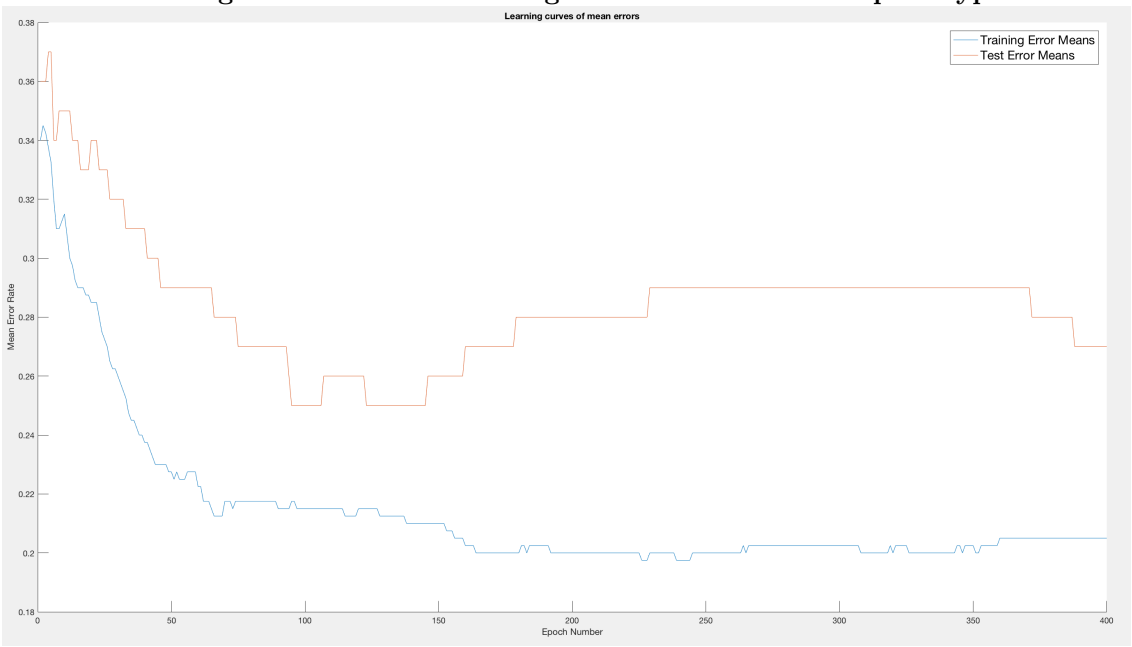

Figure 4: *shows the epoch number vs the mean training and test error rates when the algorithm is executed with 8 prototypes and a learning rate of 0.001*

We can observe that, while the training error keeps decreasing, the test error does not do the same: in the beginning it decreases as well, but, after a certain number of epochs, it increases again. This happens when the model is overfitting, i.e. it is adapting to fit as well as possible the training data, but it is not considering the underlying pattern, decreasing thus the probability of correctly classifying points, which are not in the training set. This happens when we have a complex model, i.e. a model with many prototypes, for lvq, or with many hidden units, for neural networks. That is why this particular trend in the mean test errors is more visible with 6 and 8 prototypes, while it is less evident with 2 prototypes. The more complex a model is, the more probably it overfits the training data, unless it is properly controlled through other parameters, such as the number of training epochs or the learning rate: by changing these values we can control the effective complexity, which is the degree to which the training error can be minimized. This can be clearly seen in **Figure 5** , which shows how the number of prototypes influences the mean training and test errors at the end of the whole lvq algorithm. The code in **Listing 4** shows how the plots in **Figure 5** have been obtained.

Listing 4: *Code to observe the influence of the number of prototypes on the training and test errors*

```
 % Show the influence of number of prototypes on the mean training
and test errors by plotting the former vs the latter
 final_training_errors = zeros(1, 35);
 final_test_errors = zeros(1, 35);
 for i=1:35
     [final_training_error, final_test_error] = n_fold_validation(
data, 5, 0.1, P, 2*i, 400, 1);
     final_training_errors(i) = final_training_error;
     final_test_errors(i) = final_test_error;
 end
 plot_functions(3, 35, final_training_errors, final_test_errors,["
Training Error" "Test Error"],["Number of prototypes per class" "
```

```
    Error Rate"]);
```

In **Figure 5** , while we increase the number of prototypes used, we see the training error become smaller and the test error become larger (as expected, considering the previous discussion about complexity and overfitting). For the following plot, we used a learning rate of 0.1, decreased of its 5% after each epoch, and 400 epochs: we chose the decreasing learning rate to obtain good results in less epochs, since the prototypes are updated only a few times during each epoch (especially when we get to more than 50 prototypes) and having a large beginning learning rate makes it reach good results much faster. This time, to obtain more precise and execution-independent results, we executed the algorithm 10 times for each number of prototypes, and then used the mean of the test and training errors in the following plot.

Final mean test and training errors for different numbers of prototypes

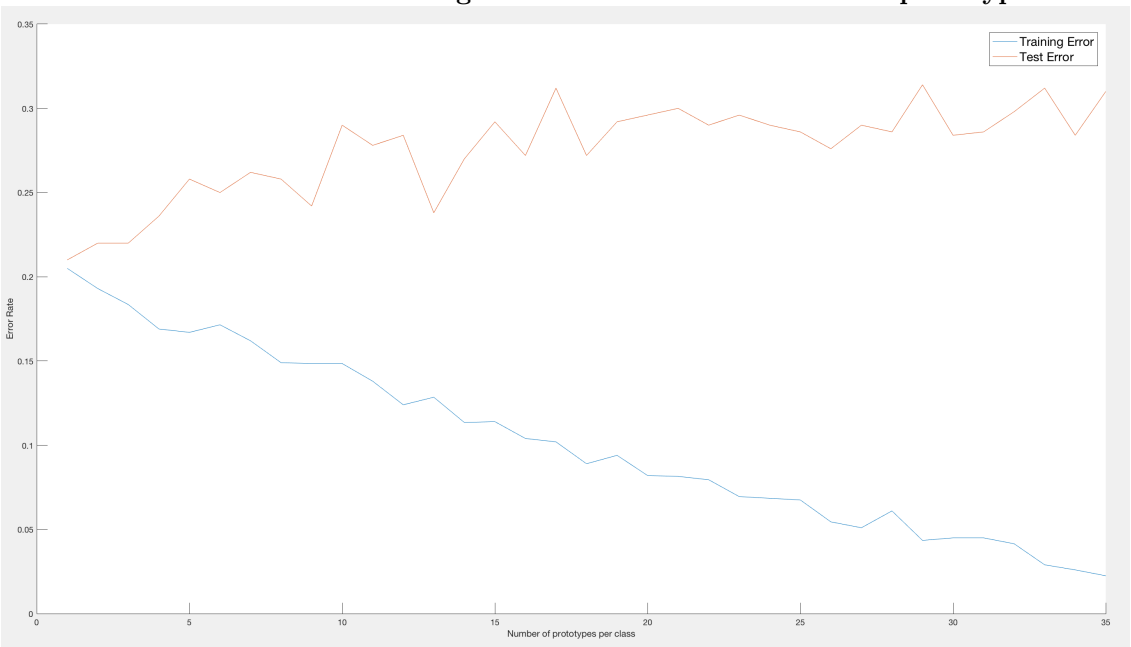

Figure 5: *shows the number of prototypes per class vs the final mean test and training errors(meaning the mean training and test errors at the end of the algorithm, not after each epoch)*

To solve the problem of overfitting we could try to change the complexity of the model based on the results of the test sets, but if we did that, we would risk overfitting also the validation procedure, thus requiring a multi-layered validation process: that is why it is usually better to tune the parameters only using the training sets, otherwise the performance, with respect to entirely new data, will be unclear.

Number of folds

We can identify two extreme cases in cross-validation: Holdout and leave-one-out validation. The former occurs when the number of "folds" is only two, dividing thus the data set in 2 disjoint subsets, while the latter occurs when the number of "folds" equals the number data examples, Using the holdout method we have as training set only half of the data, therefore we could be easily missing some important regularities and have a high biased system. By increasing the number of folds, we reduce the bias, as we are using more data for training, but we also increase the variance, as we are using less data for testing and thus we can hardly understand when the model is overfitting. The maximum folds number that we can choose is the size of the data set, leading to a leave-one-out

validation, which uses a single example as test set and all the other data as training set, for each iteration: in this case we will generally have a low bias, but we will have a high variance, because the training set will still be almost all the data set. To show this we executed the algorithm several times, with various number of folds(2,4,5,10,20,25,50,100), as shown in **Listing 5** , and plotted the final mean training and test errors in **Figure 6** . Also in this case we repeated each algorithm 10 times and considered the mean values, to obtain more precise and execution-independent results.

Listing 5: *Code to observe the influence of the number of folds on the training and test errors*

```
 % Show the influence of number of folds on the mean training and
test errors by plotting the former vs the latter
 final_training_errors = zeros(1, 8);
 final_test_errors = zeros(1, 8);
 ns = [2 4 5 10 20 25 50 100];
 i = 0;
 for n = ns
     i=i+1;
     ftre = 0; ftee = 0;
     for j = 1:10
         [final_training_error, final_test_error] =
n_fold_validation(data, n, 0.1, P, 4, 400, n);
         ftre = final_training_error + ftre;
         ftee = final_test_error + ftee;
     end
     final_training_errors(i) = ftre/10;
     final_test_errors(i) = ftee/10;
 end
 plot_functions(5, ns, final_training_errors, final_test_errors,["
Final Mean Training Error" "Final Mean Test Error"],["Number of
folds" "Error Rate"]);
```

Final mean test and training errors for different numbers of folds

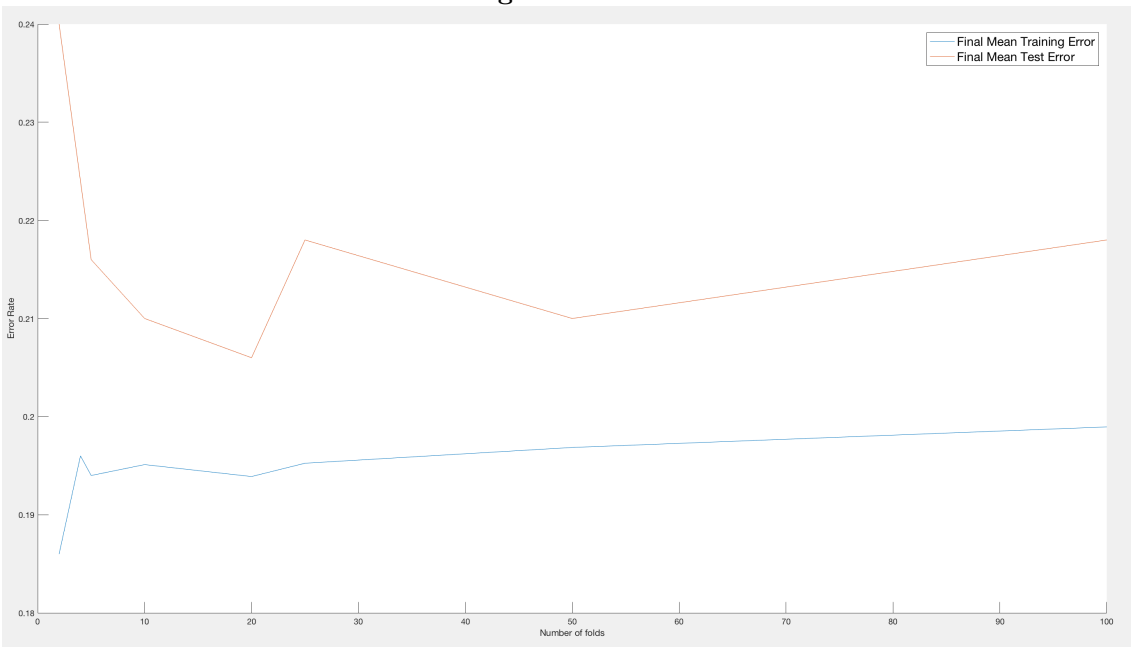

Figure 6: *shows the number of folds vs the final mean test and training errors*

Standard deviation

The standard deviation of the test errors obtained in every fold of the process can be used to determine how "stable" a model is: if we have a large standard deviation, it means that slightly changing the test set gives significantly different results, and thus that the quality of the classifier strongly depends on the training and on the test set; on the contrary, if the standard deviation is small, it means that the classifier is steady and should have the same performance, even on new unknown data. This holds only if we assume that the training data is somehow representative of the data that will be classified in the future. **Figure 7** shows the plots of the means (first row) and the standard deviations (second row) obtained executing the same algorithm with the same parameters: each column is an execution. These plots have been obtained with 4 prototypes, and considering also the plots in **Figure 8** , we can observe that, with a smaller standard deviation, we expect less difference between the test errors of the three executions, while with a slightly larger standard deviation, as in **Figure 8** , we expect more different results. Note that **Figure 8** has been obtained with 10 prototypes.

Repeated learning curve for the mean test and training errors and their standard deviation

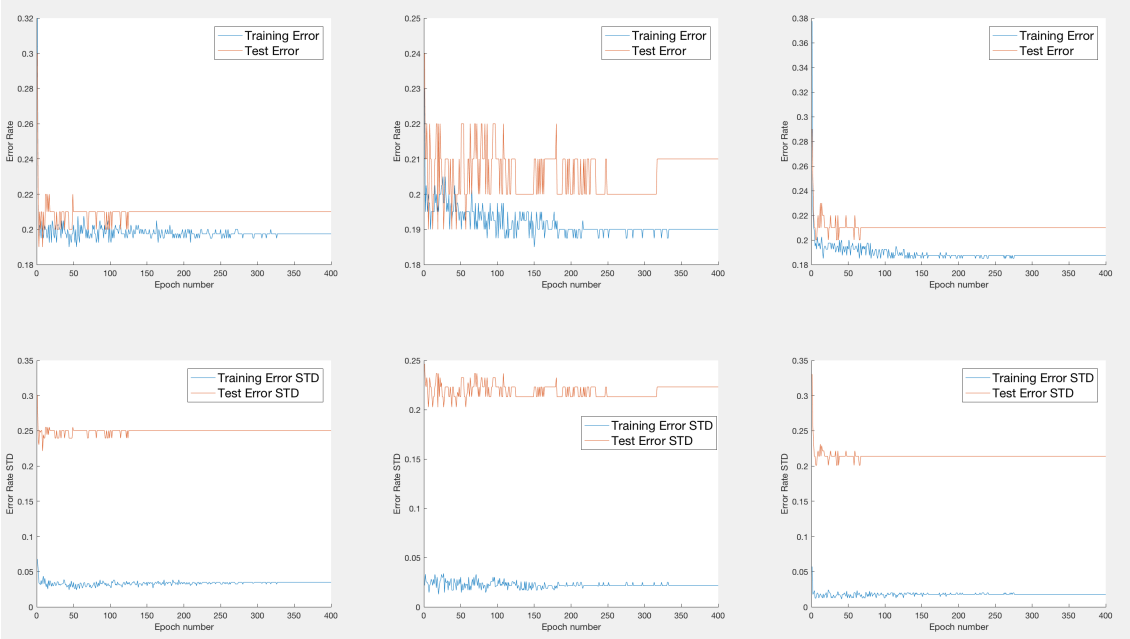

Figure 7: *shows the epoch number vs the mean test and training errors and the standard deviations from those means. In this plot 4 prototypes and a decreasing learning rate, initialized at 0.1, were used*

Repeated learning curve for the mean test and training errors and their standard deviation

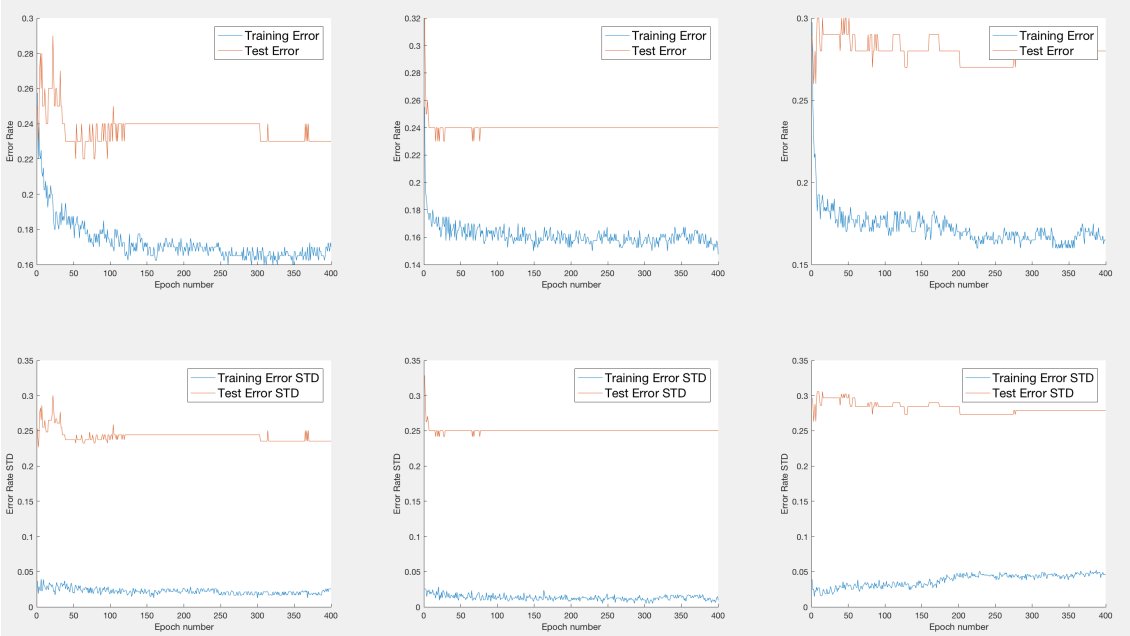

Figure 8: *shows the epoch number vs the mean test and training errors and the standard deviations from those means. In this plot 10 prototypes and a decreasing learning rate, initialized at 0.1, were used*