Introduction to Machine Learning
Lab Session 3

Group 42
Daniel Skala (s3953602) & Rares Dobre (s4051203)

October 6, 2021

Assignment 3

Introduction

In this assignment we are asked to perform a M-fold cross validation using the LVQ1 algorithm and the dataset from the first assignment, which contains 100 2-dimensional vectors, that are assigned to two different classes. Cross validation experiments are commonly used to evaluate the performance, and the generalisation capability of various machine learning algorithms using new data.

This report presents an outline of our implementation of the M-fold cross validation[1] using the LVQ1 algorithm, the results that the algorithm has achieved, accompanied by a corresponding discussion of them. Afterwards, three bonus implementations will be presented, together with our individual contributions for this assignment. The report ends with the implementation of the M-fold cross validation, which is conveniently placed under *Appendix 1.*

Method

In this section we will describe the core implementation of the algorithm, the methods we have used to run our experiments, as well as the structure of the experiments themselves. Moreover, any implementation tricks that were employed will be stated after the description of the implementation. The main idea behind the validation procedure is to randomly split the given dataset into a *training* set and a *testing* set, perform the corresponding training (using LVQ1 for example) on the *training* set, and evaluate its generalisation capability with regards to the *testing* set.

M-fold Cross-validation
The cross-validation algorithm begins by reading in the dataset, assigning the corresponding classes, and then shuffling the dataset by simply permuting the rows of the dataset. Next, the parameters of the algorithm are set, such as the number of prototypes, the number of disjoint sets, the learning rate and the maximum number of epochs. The algorithm follows up by splitting the dataset into $M = 5$ disjoint subsets, each containing 20% of the data. In each training run, we use 4 subsets for training, and one for validation. Hence, the LVQ1 algorithm will be called on those 4 subsets, with K prototypes per class. Prior to starting the LVQ1 algorithm, K prototypes from each class will be initialised by random selection from the training set. Each training will run up to 100 epochs. After the end of each training run, the *training* error and the *validation* errors are computed. To compute the errors, each example in the *training* or the *testing* set is presented, and the closest prototype is identified. If their classes do not match, an error is recorded. After each example from

[1] The M-fold cross validation will be performed using the LVQ1 and the Linear regression algorithms, algorithms which will not be discussed again

the set has been presented, the error is normalised to the size of the set. An alternative formula for computing the error can be observed below:

$$error \;=\; \frac{\#\{\; a_i \mid a_i \in exampleSet \;\wedge\; class(a_i) \neq class(w^*)\}}{\#exampleSet}$$

where *exampleSet* is either the *training* set or the *testing* set, w^* is the closest prototype to the example at hand, and # represents the cardinality operator. Note that here the error is normalised to the size of the set, such that the *training* and the *validation* errors could be compared.
This process is repeated for the $M = 5$ possible splits into the training and the validation sets, and each *training* and *validation* error is recorded. In the end, the average and the standard deviations of the errors are computed, and, if K (the number of prototypes per class) is higher than 1, the whole process will be repeated for each number in the interval $[1, K]$.

Implementation tricks
We have used a couple of implementation tricks to speed up the execution of the program, but to also make the implementation straightforward. A list of the tricks that we have used can be found below, together with the lines of code where they were implemented.

1. **Shuffling the dataset**
   To shuffle our dataset we have used the function $randperm()$. Since the labels were already attached in the third column, shuffling the dataset did not redistribute the labels - they stayed within their data points. This can be seen in lines 12 and 104.

2. **Splitting the dataset**
   To generalize our splitting algorithm for any values of M (or K), we declare the starting and ending index of the testing set as follows:
   $startIndex = (1 + ((i-1) * P/M))$;
   $endIndex = (i * P/M)$;
   where P is a number of examples (100), M is a number of splits (5) and i is our iterator which ranges from 1 to M. This way we specified the bounds for the testing set (which will be within the start and the end index). Hence, the training set will be a concatenation of the dataset from 1 to $startIndex - 1$ and from $endIndex + 1$ to the end of the dataset. This can be seen in lines $23 - 26$.

3. **Selecting prototypes**
   To select K prototypes for every class, we randomly pick a data point from the training set and check the corresponding class. This is followed by a while loop which keeps selecting new prototypes and ensures that they are from the desired class. This can be seen in lines $29 - 45$.

Results

In this section we will show the results we have obtained after running the $M - fold$ cross validation. The figure below depicts the normalised error rates for training and testing per values of K including the standard deviations. The blue line represents the training set and the orange line the testing set. Three plots will be provided, one which includes both the training error and the validation error (for comparison), and one plot for each error rate. Additionally, we have included also a plot that shows the errors compared for $K = 15$ and $t_{max} = 500$.

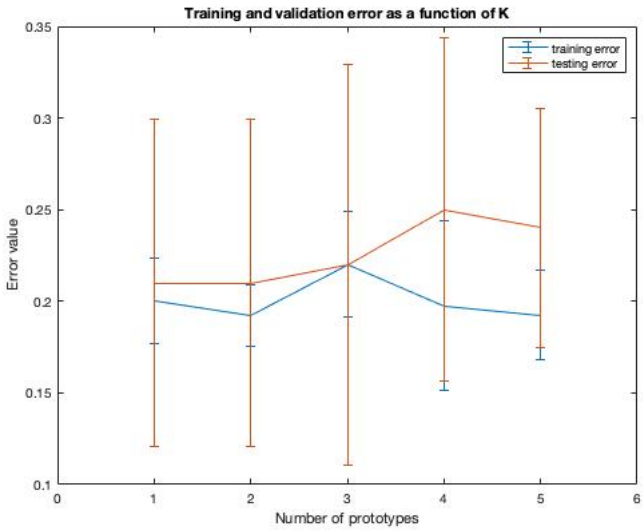

Figure 1: *Training and validation error plotted against each other*

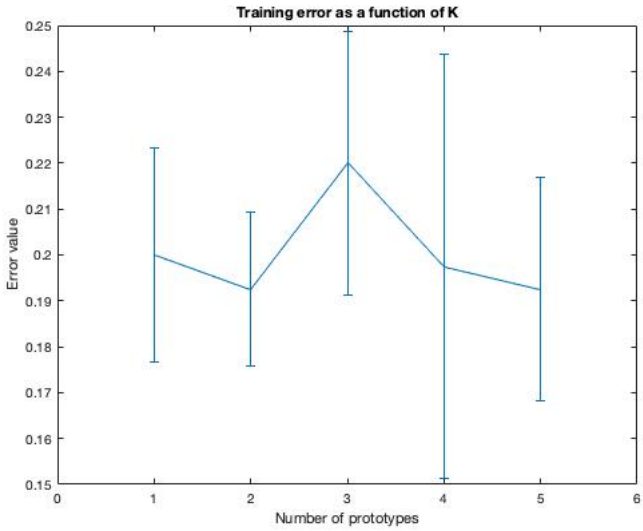

Figure 2: *Training error as a function of the number of prototypes*

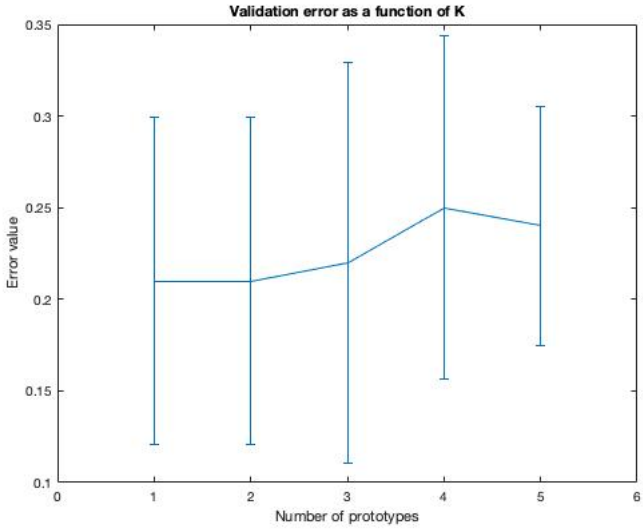

Figure 3: *Validation error as a function of the number of prototypes*

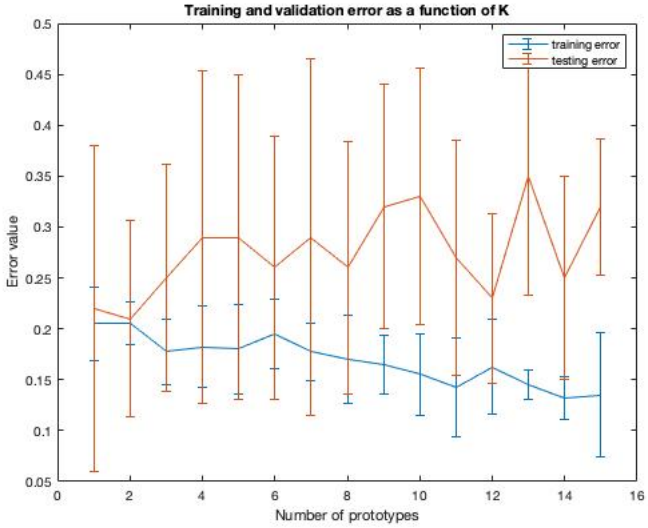

Figure 4: *Training and validation error plotted against each other for* $K = 15$ *and maximum* 500 *epochs*

Discussion

From *Figure 1* we observe that the error rates for values of K up to $K = 5$ are approximately within the range of 0.19 and 0.26 for both training and testing set (although these values tend to fluctuate a bit more for other runs). This can also be inferred from the standard deviations which are very large especially for the validation set.

The desired error rate would be decreasing with both training and testing set. However, we expect the errors for the testing set to be a bit higher since these are the values that our model has never seen before. Once the validation error rate starts increasing again while the training error rate keeps decreasing, we can conclude that our model might have over-fitted to the provided training set.

It is important to acknowledge that when we had to perform the LVQ1 training on the whole dataset, the error rate stabilised in the proximity of 0.2. Looking at the graphs for the training and validation errors, we can clearly see that using $K = 2$ prototypes per class, the error rate tends to the original rate, which is desirable.

As we add more prototypes, we increase the "complexity" of our model. This is, of course, good for the training set since the model is getting more and more used to the data it sees. This often results in decreasing error rate for the training set. However, since the model over-fitted the training data, the testing error starts increasing and our model no longer performs well for unforeseen examples.

Looking at *Figure 4*, we can clearly see the decreasing training error rate and increasing testing error rate from $K = 2$. At this point, we over-fitted our model by increasing the "complexity" of the model in terms of adding more prototypes. This means that to get optimal results, we should stop the training at $K = 2$ because the testing error is minimal. We should also note that when the number of prototypes is very small, the *bias* of our model is too large while the *variance* is low. On the contrary, for a high number of prototypes (eg. $K = 10$) the *variance* is high and the *bias* is low. A good compromise between these two factors is when the number of prototypes is approximately 2.

There are multiple ways to prevent over-fitting. In the case of the LVQ1 implementation, one could also reduce the maximum number of epochs of training, or the learning rate. We have found that limiting the amount of prototypes per class is the most effective procedure in this case.

Looking at the (large) standard deviations of the error rates, namely for the testing set, it might not be clear at which value of K we should stop our training. We might have to run M-fold cross validation for many values of K multiple times, average the results and then we might get a more precise indication of when exactly our model over-fitted. As this takes a lot of compute time, we have decided to skip this additional validation step.

Bonuses

10-fold cross validation

For this bonus we have implemented 10-fold cross validation (which is more commonly used in practice) by incrementing M to 10. By increasing the value of M, we increase the size of the training set and we decrease the size of the testing set. This way, more and more examples will be taken into consideration whilst training, however, the generalisation power will only be tested according to less examples.

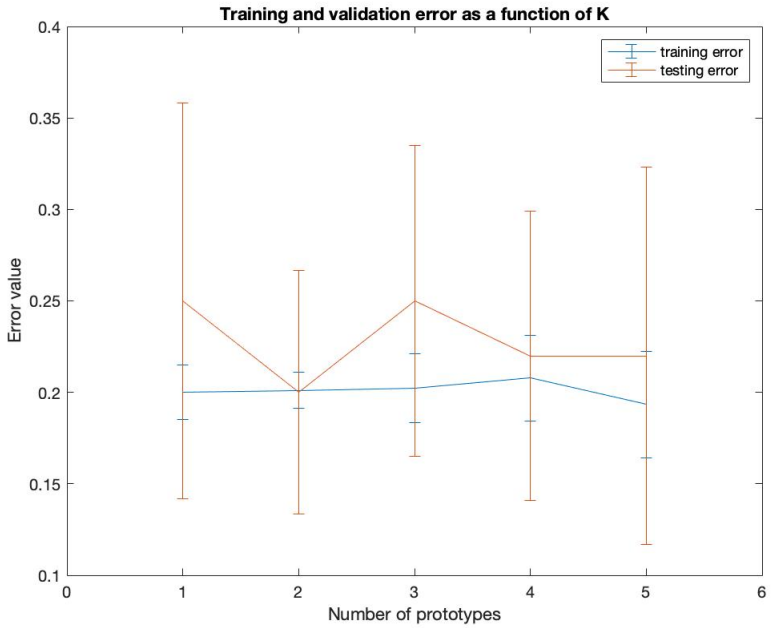

Figure 5: 10-*fold cross validation for* $K = 10$*, averaged + normalised error rates*

As we can observe from the graph, the training error is mostly constant, with low fluctuations around $K = 4$ and $K = 5$ prototypes. The almost constant training error can be seen as a consequence of the size of the training set. For each *trainingSet* and *testingSet* split, the *trainingSets* will highly overlap with one another, leading to almost the same error rate. Overfitting can also be observed here when $K \geq 3$, however, it is not as severe as the previous cases when $K = 15$ for example.

Experimenting with the different learning rates

The results that have been given in the **Results** section use a learning rate equal to 0.002. In this bonus, we experiment with various learning rates, and we use $K = 1$ for efficiency purposes. To find an approximation of the "best" learning rate, one could easily take an interval, say $[0.1, 0.00001]$, evaluate the errors at the interval limits, and then intuitively shrink the interval. In other words, if the error rate for a learning rate of 0.1 is too high, we know that it must be smaller. In this bonus, we will showcase and analyze the impact of five learning rates, namely 0.2, 0.1, 0.01, 0.001, 0.0001. The error rates corresponding to each learning rate can be found below:

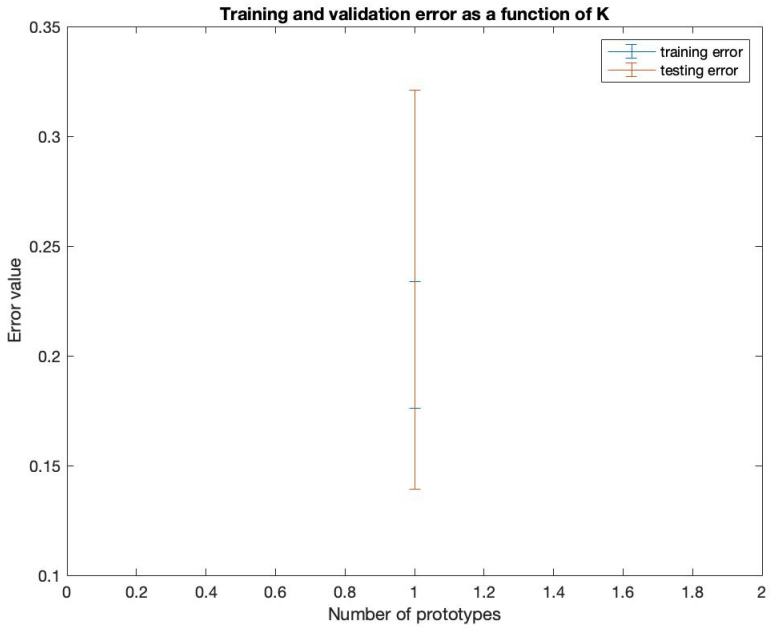

Figure 6: *Training and validation errors for* $learningRate = 0.2$

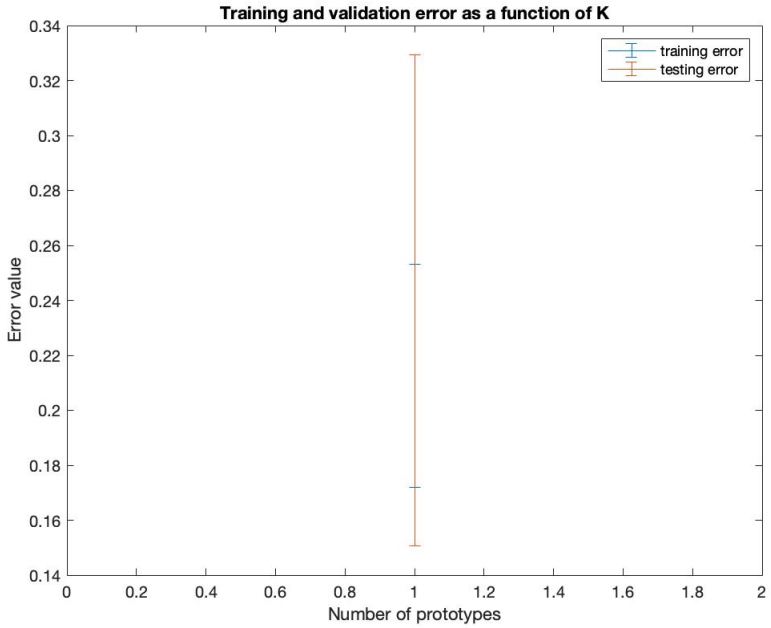

Figure 7: *Training and validation errors for* $learningRate = 0.1$

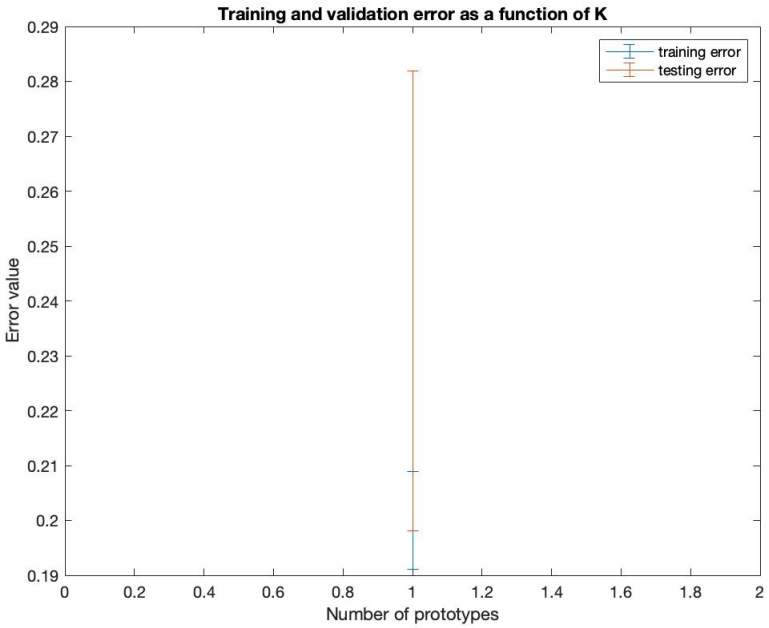

Figure 8: *Training and validation errors for* $learningRate = 0.01$

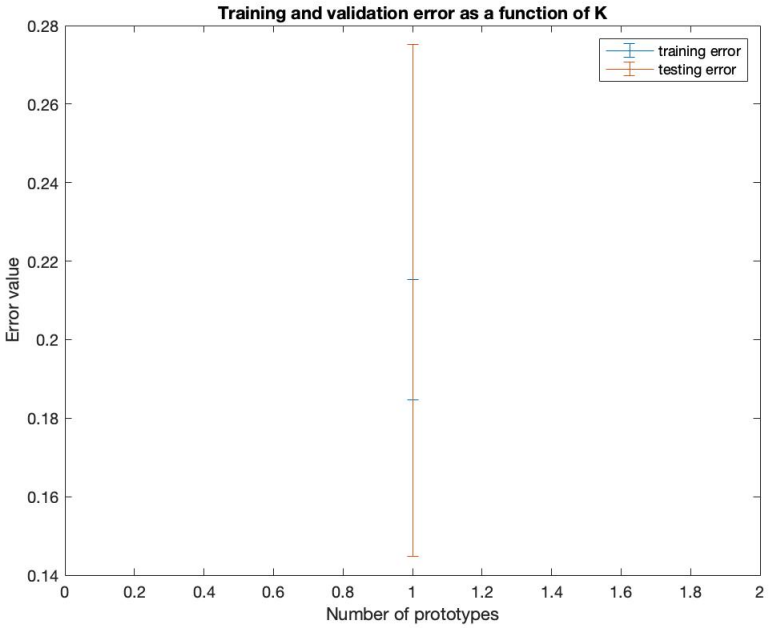

Figure 9: *Training and validation errors for* $learningRate = 0.001$

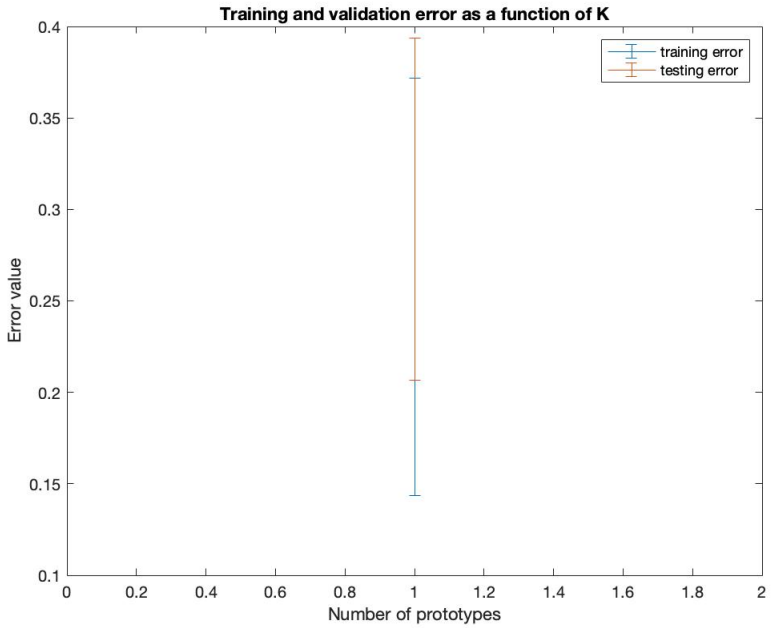

Figure 10: *Training and validation errors for learningRate* = 0.0001

From the graphs depicted above, we can clearly see that a learning rate around 0.2 is way too big, while a learning rate of 0.0001 is too small. We conclude that the *best* learning rate is approximately 0.001 or 0.002 (as we have used in the **Results** section).

M-fold cross validation for the regression problem

To implement M-fold cross validation for the regression problem and dataset, minor tweaks had to be made to the regression implementation. First, we started by aggregating the testing and the training sets into a single set, which is permuted randomly afterwards. Next, we perform the classic split into a *training* and a *testing* set, and compute an estimate from the *testingSet*. Using that estimate, we compute the mean squared errors for the validation and training sets, and plot them as functions of the current split.

From the graph depicted below, we can observe that the MSE for the validation and for the training sets are close to the result achieved without any validation (an MSE = 0.5). Since the split was done using $M = 5$ disjoint sets, each training run will be done with 80% of the data, allowing for an even better approximation of the estimate w^*, meaning that it will get even closer to the true vector w^*. Increasing the size of the validation set and decreasing the size of the training set would result in a smaller training error, but a larger validation error.

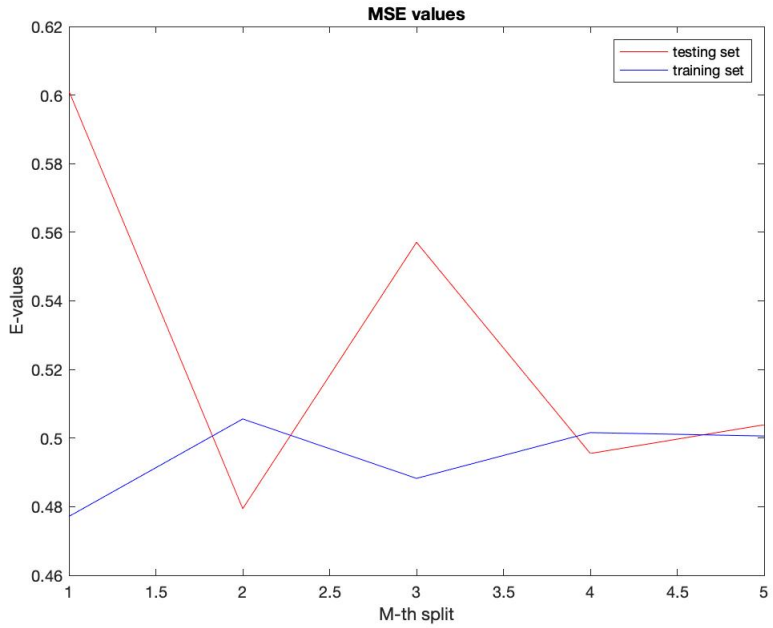

Figure 11: *MSE errors for the training and validation sets as functions of the current split*

Individual contributions

The practical was done concurrently, via multiple online sessions where we have discussed our approaches and used pair programming to deal with implementing different aspects of the code. We have used Discord to facilitate this. Hence, the program was designed and implemented by both of us concurrently, so the percentages would be 50% and 50%. Regarding answering the questions posed and writing the report, we have used Overleaf so that both of us can work at the same time on the report and we have divided the questions evenly. Hence, for writing the report and answering the given questions, our contributions were again 50% and 50%.

A. Assignment 3

Listing 1: *Code for lvq evaluation*

```
% Load the dataset
data = load("lvqdata.mat").lvqdata;

% Assign the labels
data(1:50, 3) = 1;
data(51:100, 3) = 2;

N = 2; % Dimensionality of the input vectors
P = 100; % Number of examples

% Shuffle the data
data = data(randperm(P), 1:3);
M = 5;
K = 5;
learning_rate = 0.002;
t_max = 100;
testingSet = [];
trainingSet = [];
avg_errors = [];
for k = 1:K % Try more prototypes per run
    errors = [];
    for i = 1:M % For every division of the training and testing set
        startIndex = (1 + ((i-1) * P/M));
        endIndex = (i * P/M);
        testingSet = data(startIndex:endIndex, :);
        trainingSet = [data(1:(startIndex-1), :); data((endIndex+1):P,
    :)];

        % Generate prototype for class 1
        prototypes = [];
        for prot = 1:k
            selectedPrototype = trainingSet(randi(P - P/M), :);
            while selectedPrototype(1, 3) ~= 1
                selectedPrototype = trainingSet(randi(P - P/M), :);
            end
            prototypes(prot, 1:3) = selectedPrototype(:, :);
        end

        % Generate prototypes for class 2
        for prot = 1:k
            selectedPrototype = trainingSet(randi(P - P/M), :);
            while selectedPrototype(1, 3) ~= 2
                selectedPrototype = trainingSet(randi(P - P/M), :);
            end
            prototypes(prot + k, 1:3) = selectedPrototype(:, :);
        end
```

```
        prototypes = lvq1(prototypes, 2*k, t_max, (P - P/M),
    trainingSet, learning_rate);
        trainingError = computeErrors(prototypes, 2*k, (P - P/M),
    trainingSet);
        testingError = computeErrors(prototypes, 2*k, P/M, testingSet);

        errors(i, 1) = trainingError;
        errors(i, 2) = testingError;
    end
    avg_errors(k, 1) = mean(errors(:, 1)); % Training error
    avg_errors(k, 2) = std(errors(:, 1)); % Training standard
    deviation
    avg_errors(k, 3) = mean(errors(:, 2)); % Testing error
    avg_errors(k, 4) = std(errors(:, 2)); % Testing standard deviation

end

errorbar(1:K, avg_errors(1:K, 1), avg_errors(1:K, 2));
title('Training error as a function of K');
xlabel('Number of prototypes');
ylabel('Error value');
xlim([0 (K+1)]);
figure;
errorbar(1:K, avg_errors(1:K, 3), avg_errors(1:K, 4));
title('Validation error as a function of K');
xlabel('Number of prototypes');
ylabel('Error value');
xlim([0 (K+1)]);
figure;
errorbar(1:K, avg_errors(1:K, 1), avg_errors(1:K, 2));
hold on;
errorbar(1:K, avg_errors(1:K, 3), avg_errors(1:K, 4));
hold off;
legend("training error", "testing error");
xlim([0 (K+1)]);
title('Training and validation error as a function of K');
xlabel('Number of prototypes');
ylabel('Error value');

function computedError = computeErrors(prototypes, K, P, data)
    computedError = 0;
    for i = 1:P
        example = data(i, 1:3);
        winning_prototype_idx = 0;
        for j = 1:K
            if winning_prototype_idx == 0 || pdist2(example(1, 1:2),
    prototypes(j, 1:2), "squaredeuclidean") < min_distance;
                winning_prototype_idx = j;
                min_distance = pdist2(example(1, 1:2), prototypes(j,
    1:2), "squaredeuclidean");
            end
```

```
        end
        if prototypes(winning_prototype_idx, 3) ~= example(1, 3)
            computedError = computedError + 1;
        end
    end
    computedError = computedError / P;
end

% The LVQ1 Algorithm
function prototypes = lvq1(prototypes, K, t_max, P, data,
   learning_rate)
    for t = 1:t_max
        epoch_errors = 0;
        randomised_data = data(randperm(P), 1:3); % Randomly permute
   the data including the third column
        for i = 1:P
            example = randomised_data(i, 1:3);
            % Find the nearest neighbor
            winning_prototype_idx = 0;
            for j = 1:K
                if winning_prototype_idx == 0 || pdist2(example(1,
   1:2), prototypes(j, 1:2), "squaredeuclidean") < min_distance;
                    winning_prototype_idx = j;
                    min_distance = pdist2(example(1, 1:2), prototypes(
   j, 1:2), "squaredeuclidean");
                end
            end
            % At this point we have the nearest point to the prototype
            if prototypes(winning_prototype_idx, 3) == example(1, 3)
                psi_flag = 1;
            else
                psi_flag = -1;
            end
            % The update step
            prototypes(winning_prototype_idx, 1:2) = prototypes(
   winning_prototype_idx, 1:2) + learning_rate * psi_flag * (example
   (1, 1:2) - prototypes(winning_prototype_idx, 1:2));
        end
    end
end
```