Faculty of Mechanical Engineering and Robotics

Department of Robotics and Mechatronics

Basics of AI and Deep Learning
Course for Mechatronic Engineering with English as instruction language

Instruction 2:

Linear and nonlinear regression, data storage and display

You will learn: How to implement from scratch and configure basic linear and nonlinear regression algorithms. In addition to that you will also learn how to store and display experimental data efficiently.

Additional materials:

- Course lectures 1 and 2 [*obligatory*]

Learning outcomes supported by this instruction:
[Here a list of learning outcomes' codes]

Course supervisor:
Ziemowit Dworakowski, zdw@agh.edu.pl

Instruction author:
Ziemowit Dworakowski, zdw@agh.edu.pl

Linear and polynomial regression

Lets start by loading and showing a dataset for 2-dimensional regression problem. The datasets on which we'll be working in this task are in the *RegressionData* folder. I'll start with dataset *DataForDemonstration*, which after loading and displaying with the following code looks like in Fig 1:

```
load RegressionTrainingData


figure;
plot3(RegressionTrainingData(:,1), RegressionTrainingData(:,2), RegressionTrainingData(:,3),'.r')
xlabel('x1')
ylabel('x2')
zlabel('value')
grid on
```

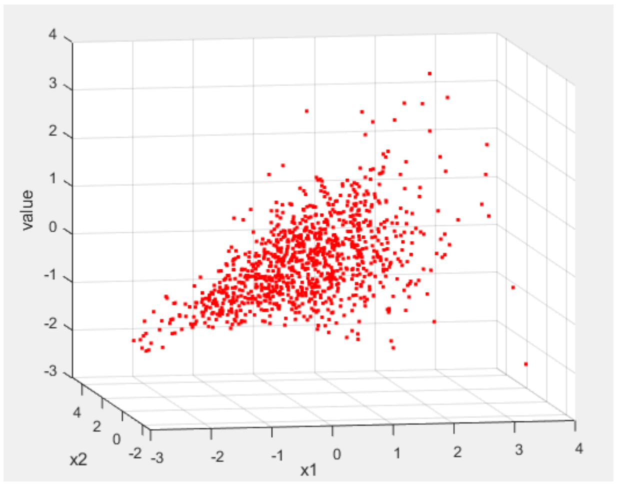

Figure 1 – Sample dataset

We will treat 3rd data column as a target value which we should learn to predict. Now our goal is to build a regressor. We need to somehow find the coefficients of the regression plane – a model that allows for prediction of new samples. We will do it by means of optimization. Our objective function will be the error between predicted values and actual values and we will find the solution using algorithms developed last time.

Lets start with this simple model:

```
function [PredictedValue] = LinearRegressionModel(X,W)
        PredictedValue = W(1)*X(1) + W(2)*X(2) + W(3);
end
```

Note that this is a function (save it as a separate file) which takes vector of features X = [x1,x2] and vector of parameters W = [W1, W2, W3] governing how the actual model works. Lets fix the vector of parameters at any arbitrary values and lets just pass all the training data through the model to see what happens:

```
W1 = 1;   % Since we don't know what the model parameters should be – lets start with ones.
W2 = 1;
W3 = 1;

for k = 1:length(RegressionTrainingData)
    PredValue(k) = LinearRegressionModel([RegressionTrainingData(k,1),...
RegressionTrainingData(k,2)],[W1,W2,W3]);
    % The third column of TrainingData is our target
    Error(k) = PredValue(k) - RegressionTrainingData(k,3);

end

MSE = mse(RegressionTrainingData(:,3),PredValue');
```

Now we can plot error values themselves or show how far we are from the actual points – see result in Fig. 2:

```
figure;
subplot(1,2,1)
  plot3(RegressionTrainingData(:,1),RegressionTrainingData(:,2),RegressionTrainingData(:,3),'.r'); hold on
  plot3(RegressionTrainingData(:,1),RegressionTrainingData(:,2),PredValue,'.k');
  xlabel('x1');  ylabel('x2')
  zlabel('value')
  grid on
subplot(1,2,2)
  plot(Error)
  xlabel('Sample number');  ylabel('Error value')
```

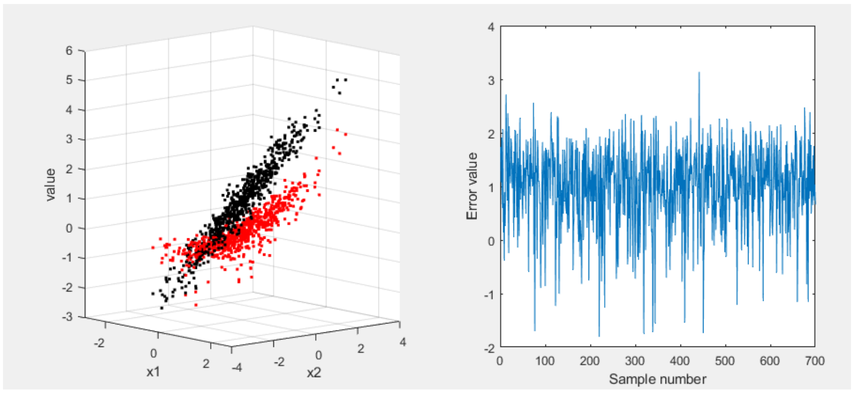

Figure 2 – Regression error in feature space (left) and shown for all the samples (right).

It is a good starting point for building an optimizer. We'll just need to put all the previous pieces of code in a function which will be our *FunctionForOptimization*. If *FunctionForOptimization* takes a vector as input, it will easily be usable with optimization algorithms developed within laboratory 1. <u>Note, that if the function is expected to run quickly, you should probably turn all the visualizations off (that means: no plotting).</u>

Task 2.1: Load your **individual*** set of training data (*RegressionTraining_n* where *n* is your individual task number). Create a new function, name it e.g. *LinearRegressorCheck.* Let the function take a vector of parameters W = [W1,W2,W3] and return MSE value for all the points in the *RegressionTrainingData* dataset. Evaluate the function for several different values of W1, W2 and W3, try to find one that is reasonably good.

Task 2.2: Use your *LinearRegressorCheck* as function for optimization with your grid search algorithm developed in scope of laboratory 1. Find the optimum and store it in Table 2.1.

Task 2.3: Use your *LinearRegressorCheck* as function for optimization with your 1+1 algorithm with adaptive step developed in scope of laboratory 1. Find the optimum and store it in Table 2.1.

Task 2.4: Use your *LinearRegressorCheck* as function for optimization with your multistart gradient algorithm with adaptive step developed in scope of laboratory 1. Find the optimum and store it in Table 2.1.

Task 2.5: Modify your regression model, to allow for more complex geometry. Let it be a polynomial quadratic model given by equation:

$$y = w_1x_1 + w_2x_2 + w_3x_1x_2 + w_4x_1^2 + w_5x_2^2 + w_6$$

Optimize this 6-dimensional model using 1+1 algorithm with adaptive step and store the results in Table 2.1 at the end of the instruction. Try to configure the method in such a way that they consistently find global minimum. Note that you will probably need to use more optimization steps to find solution in a 6-dimensional feature space.

Task 2.6: Repeat task 2.5, this time using a multistart gradient model. Start with picking the value of objective function checks equal to the one used in task 2.5 and now configure the number of starts and iterations of your multistart gradient to aim for the most consistent (and best) result. Then store the result in table 2.1.

Task 2.7: So far we used only *Training* datasets. Now time to evaluate our models on data which were not seen during training. Lets use the learned parameters to check the performance of your models on the *Testing* data subset and store the results in table 2.1.

Table 2.1: Regression results

	W1	**W2**	**W3**	**MSE**			**MSE validation**
Manual setup 1							-
Manual setup 2							-
Manual setup 3							-
Grid search							
1+1 method				Try 1:	Try 2:	Try 3:	
Gradient method				Try 1:	Try 2:	Try 3:	
1+1: polynomial model	-	-	-	Try 1:	Try 2:	Try 3:	
gradient: polynomial model	-	-	-	Try 1:	Try 2:	Try 3:	

Storing and showing statistical data from multiple experiments in a controlled manner

Up until now we were storing our results in tables (probably handwritten in your notebook or manually annotated onto a pdf instruction file). This is not the optimal way, though. From now on, lets use data structures for that. Lets say that we have an experiment in which we repeat a test of some algorithm's configuration several times and then we do the same for a different configuration. We would like to have these results stored in a tree structure that would look like in Figure 6.

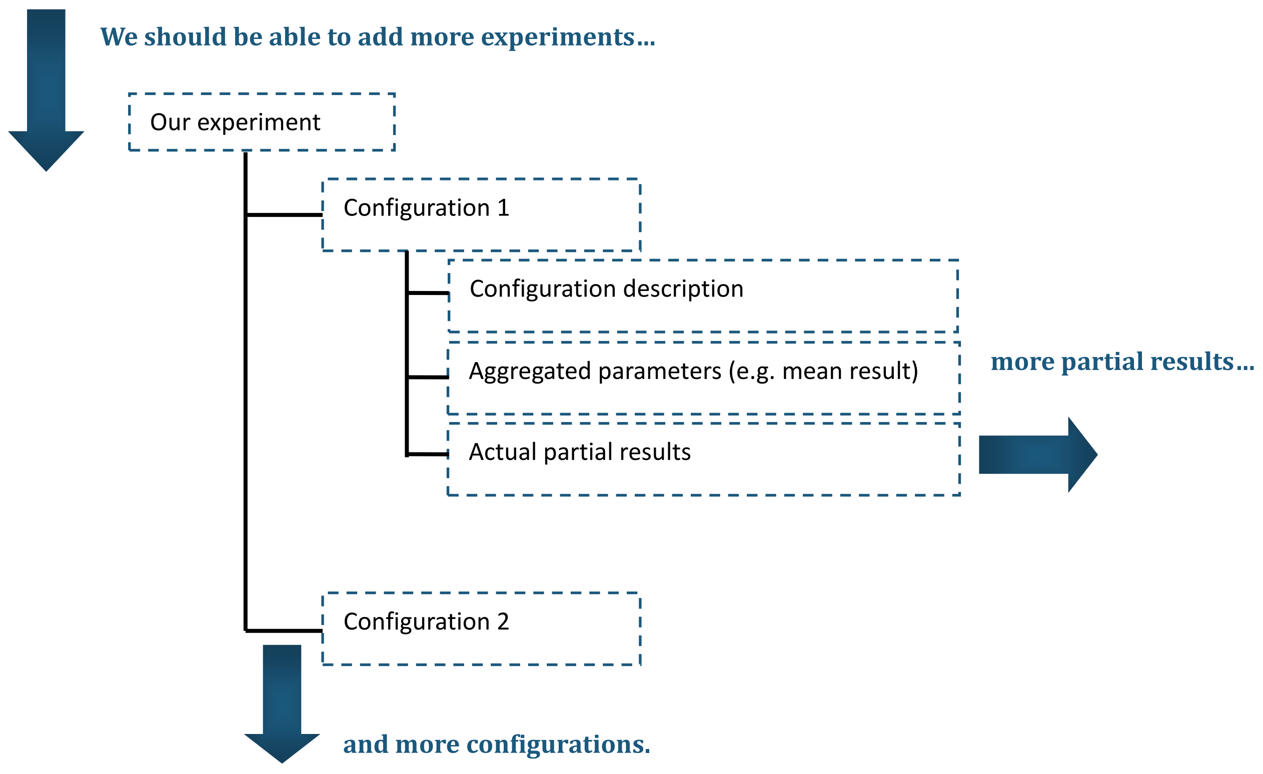

Figure 6 – Data structure for organization of experimental results

For the sake of explanation lets imagine that we are working again on the first instruction and we are testing optimization algorithms in optimization of our artificial 2D and 4D functions. Remember how we got a vector of results, mean and standard deviation in task 1.8? Lets now use that to illustrate how we could store these outcomes.

We knew that we had two algorithms to evaluate (multistart gradient and 1+1) and three functions on which we did our testing (fewminima, manyminima, multidimensional)? We should also be able to test different configurations of algorithms, that is: different metaparameter values. The easiest way to define a nested data structure to store these results is to just throw the data onto the level we want it to be – and the rest of the structure will be organized accordingly:

```
for repetition = 1:20

   %% Here we have a test which end in one Result that is stored in the Results vector:
   %% ...
   Results(repetition) = BestHistory(end)

end
% Here we calculate statistics from our run:
mean(Results)
std(Results)

% Here we store all the necessary information into the data structure:

AggregatedResults.MultistartGradient.Fewminima(1).Results = Results;
AggregatedResults.MultistartGradient.Fewminima(1).Mean = mean(Results);
AggregatedResults.MultistartGradient.Fewminima(1).Std = std(Results);
AggregatedResults.MultistartGradient.Fewminima(1).Config.Starts = 5;
AggregatedResults.MultistartGradient.Fewminima(1).Config.MaxSteps = 100;
...
```

If we then want to test another optimization method, say: to 1+1, we could just change the relevant structure's level:

```
AggregatedResults.OnePlusOne.Fewminima(1).Results = Results;
AggregatedResults.OnePlusOne.Fewminima(1).Mean = mean(Results);
AggregatedResults.OnePlusOne.Fewminima(1).Std = std(Results);
AggregatedResults.OnePlusOne.Fewminima(1).Config.InitialStep = 7;
AggregatedResults.OnePlusOne.Fewminima(1).Config.ReductionFactor = 0.8;
AggregatedResults.OnePlusOne.Fewminima(1).Config.AdaptationTrigger = 3;
```

If we decide we want to run the same algorithm for the same problem but for a different configuration (different values of metaparameters), we can just add another field in a Fewminima vector:

```
AggregatedResults.OnePlusOne.Fewminima(2).Results = Results;
AggregatedResults.OnePlusOne.Fewminima(2).Mean = mean(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Std = std(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Config.InitialStep = 4;
AggregatedResults.OnePlusOne.Fewminima(2).Config.ReductionFactor = 0.6;
AggregatedResults.OnePlusOne.Fewminima(2).Config.AdaptationTrigger = 6;
```

You can also attach structure's parts together. So the following code is equivalent to the former one:

```
Config.InitialStep = 4;
Config.ReductionFactor = 0.6;
Config.AdaptationTrigger = 6;
AggregatedResults.OnePlusOne.Fewminima(2).Results = Results;
AggregatedResults.OnePlusOne.Fewminima(2).Mean = mean(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Std = std(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Config = Config;
```

The only issue we are missing is the fact that our structure will be forgotten once we close the code – and we actually want to have just one data structure for all the codes we are running. For this reason we need a way to load existing data structure and save it afterwards:

```
if(exist('AggregatedResults'))
    load AggregatedResults
else  % If the structure does not exist - it will just be generated and saved at the end of our code
end


%% Here we have an actual method (maybe some tests, repetitions, etc.)
% ...
% ...
% ...



% Here we are storing the results of our code:
Config.InitialStep = 4;
Config.ReductionFactor = 0.6;
Config.AdaptationTrigger = 6;
AggregatedResults.OnePlusOne.Fewminima(2).Results = Results;
AggregatedResults.OnePlusOne.Fewminima(2).Mean = mean(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Std = std(Results);
AggregatedResults.OnePlusOne.Fewminima(2).Config = Config;

% Here we save the data structure with new contents:
save AggregatedResults AggregatedResults
```

How to show contents of your data structure?

If you have your data already stored in your structure, you can easily show its parts, e.g. like that:

```
plot(AggregatedResults.OnePlusOne.Fewminima(1).Results,'r'); hold on
plot(AggregatedResults.MultistartGradient.Fewminima(1).Results,'b'); hold on
legend('OnePlusOne','MultistartGradient')
xlabel('Run number')
ylabel('Result')
```

Or like that:

```
VectorOfResults = [AggregatedResults.OnePlusOne.Fewminima(1).Mean,…
AggregatedResults.OnePlusOne.Fewminima(2).Mean,…
AggregatedResults.OnePlusOne.Fewminima(3).Mean,…
AggregatedResults.OnePlusOne.Fewminima(4).Mean]
bar(VectorOfResults); hold on
```

Or just display it in the command window:

```
AggregatedResults.OnePlusOne.Fewminima(1).Mean
AggregatedResults.MultistartGradient.Fewminima(1).Mean
```

Task 2.8: Go back through the instruction and prepare your way of storing data from multiple experiments. Then do statistical evaluation of linear and polynomial models for regression (tasks 2.3, 2.4, 2.5 and 2.6). You will need to run each task at least 20 times for that. Your data structure should contain:

- Information about the solved task type (classification or regression)
- Information about the classifier used
- Information about metaparameters of the classifier used
- Statistical estimates calculated for multiple runs (mean and standard deviation)
- Source results (results of each run)
- Information about validation efficiency calculated in task 2.6 for the regression models

Ways of displaying statistical and comparison data

Once you have your data prepared, it is time to show them in a way which is easy to comprehend. This time I won't provide you with exact code to do this task – you will need to work your way thorough this problem using matlab documentation. Lets start with your data structure (prepared in scope of task 2.7) and then compare different algorithms and different setups using plots and bar graphs. A short explanation of function syntax can be found using matlab *help* command (e.g. "help bar"). Full documentation including examples of use is available after using *doc* command. Here is a list of functions you should consider:

bar allows to build bar graphs (also multicolor and comparing different vectors)
subplot allows to provide many different plots on one figure
plot3 allows to plot 3D scatterplots
line allows for plotting lines in your graphs and plots (e.g. indicating level of something in comparison to something else)

Task 2.9: Show data from laboratory 2 using visualizations designed by you. Let your figures have legends, axes labels and different colors for showing different values. Make them as professional-looking and custom as you can. Include at least:

- Comparison of mean validation and training results from task 2.7 using bar graphs for three different models – on three subplots in one figure
- Display of a source results for statistics (results of each of the 20 runs for the model) along with a line indicating mean value

Additional tasks:

Task 2.10: Modify task 2.5 so it would use locally weighted regression instead of a polynomial model. Then compare the results obtained on *validation* dataset with ones from relevant ones from previous tasks. Note that in order to get prediction of points in your validation data you should use training data (so *validation* dataset provides only points to check. Points into which you are fitting your lines should still come from *training*)

Task 2.11: Modify task 2.6 so it would use a higher order of a polynomial. Configure the training method, fit this polynomial into your *training* data and then evaluate the efficiency on a *validation* data. Compare 1+1 and gradient solutions as optimization solvers. Is there a similar difference than before or do we have a change in relative results?