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 3:

Line and multiline classification

You will learn: How to design your own linear classifier from scratch – and how to extend it to a nonlinear one by adding layered line-based decision. We'll also test viability of optimization algorithms in this multi-dimensional problem.

Additional materials:

- Course lectures 1 and 2 and 3

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 classification

Similarly as before, we need to have a dataset. This time, we'll generate it from a predefined cluster distribution. I'll use the following code for that:

```
load Clusters_10

Samples = 1000;          % How many data samples there are?
DataDivision = 0.5;      % How many data samples fall into which class?
v = 2;                   % v parameter of T Student's distribution

% Definition of data
for k = 1:Samples
   if(rand()>DataDivision)
       DATA(1,k) = 1;
       Ind = randi(Clusters.ClustersA);
       DATA(2,k) = Clusters.ACoordinates(1,Ind)+random('T',v)*0.15;
       DATA(3,k) = Clusters.ACoordinates(2,Ind)+random('T',v)*0.15;
   else
       DATA(1,k) = 0;
       Ind = randi(Clusters.ClustersB);
       DATA(2,k) = Clusters.BCoordinates(1,Ind)+random('T',v)*0.15;
       DATA(3,k) = Clusters.BCoordinates(2,Ind)+random('T',v)*0.15;
   end
end

for k = 1:Samples
    if(DATA(1,k) == 1)
        plot(DATA(2,k),DATA(3,k),'ok'); hold on
    else
        plot(DATA(2,k),DATA(3,k),'xb'); hold on
    end
end
xlabel('x');
ylabel('y');
ylim([-3 4])


save DATA DATA
```

Clusters of data are available in our *StudentToolbox* archive. Note that *DataDivision* value determines split of data into classes (we can have more examples in one class than in the other) while ***v*** value determines how heavy are tails of the data distribution – the lower the parameter, the heavier are distribution's tails (we are using T-Student's distribution here). These parameter values allowed for obtaining dataset visible in Fig 1. It is worth noting that using T-Student distribution instead of a gaussian allowed for generation of a bunch of outliers and conveniently connected clusters to each other while still maintaining separate cluster centers, so we've obtained a nice and interesting two-dimensional classification problem.

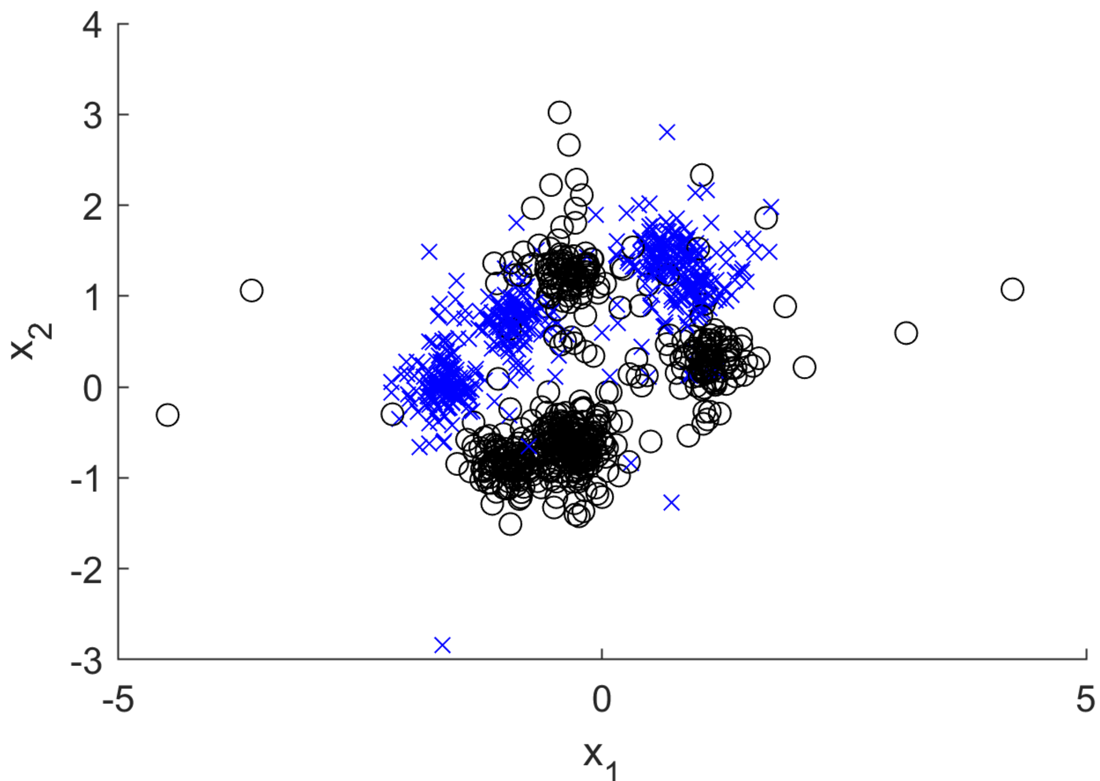

Figure 1 – A classification problem generated from Clusters_10 distribution. Note that every time you generate data, you'll get a clusters that look a bit different – but their centers should be in the same spots.

Training and testing data

Our data should now be prepared for the purpose of training and testing classifiers. We'll randomly divide it into three subsets: training, validation and testing, with 50%, 25% and 25% of data samples, respectively. In our case it would equal 500, 250 and 250 samples. To this end we can use the following code:

```
Indices = randperm(length(DATA));
DATA_permutated = DATA(:,Indices)

TR_number = ceil(length(DATA)*0.5);
VA_number = ceil(length(DATA)*0.25);
TE_number = ceil(length(DATA)*0.25);

TR_DATA = DATA_permutated(:,1:TR_number);
VA_DATA = DATA_permutated(:,TR_number+1:TR_number+VA_number);
TE_DATA = DATA_permutated(:,TR_number+VA_number+1:end);

save TR_DATA TR_DATA
save VA_DATA VA_DATA
save TE_DATA TE_DATA
```

Task 3.1: Using your **individual*** *Clusters* structure generate dataset containing 1000 samples divided equally among two classes, using *v=2.* Save generated dataset on disk so it could be used to train and test classifiers. Save both the original dataset (DATA) and its divided subsets. Save also a script that was used to generate data – so it could later be used to generate other datasets as well.

Design of a simple linear classifier

We've made sure that our dataset is not linearily separable. Nontheless, we will try and divide these data using a straight line. To design a classifier we'll just ask a simple question regarding each data point: "Is this point above or below a predefined line?"

Lets formulate equation for it:

$$W1 * x1 + W2 * x2 + b > 0 \quad (1)$$

And now lets find such *W1* and *W2*, to maximize efficiency of classification. Our classifier should look like this:

```
function [ClassLabel] = InitialClassifier(x,y,Parameters)
    if(Parameters.W1*x + Parameters.W2*y + Parameters.B > 0)
        ClassLabel = 1;
    else
        ClassLabel = 0;
    end
end
```

Such a classifier can be saved as a function and then used to classify our data as in here:

```
load VA_DATA

Parameters.W1 = 1;
Parameters.W2 = 0.3;
Parameters.B = 1;

ErrorsA = 0;
ErrorsB = 0;

for k = 1:length(VA_DATA)
   if(InitialClassifier(VA_DATA(2,k),VA_DATA(3,k),Parameters) == 1)
      % Data point classified as A
      if(VA_DATA(1,k) == 1)
          % Data point classified correctly!
          plot(VA_DATA(2,k),VA_DATA(3,k),'ok'); hold on
      else
          plot(VA_DATA(2,k),VA_DATA(3,k),'xr') ; hold on
          ErrorsA = ErrorsA + 1;
      end
   else
       % Data point classified as B
       if(VA_DATA(1,k) == 0)
        % Data point classified correctly!
          plot(VA_DATA(2,k),VA_DATA(3,k),'xk'); hold on
       else
          plot(VA_DATA(2,k),VA_DATA(3,k),'or') ; hold on
          ErrorsB = ErrorsB + 1;
       end
   end
end
xlabel('x');
ylabel('y');
ErrorsA
ErrorsB
ErrorsA+ErrorsB
```

The above script rendered results that are shown in Figure 2. There were 76 errors in A class and 27 errors in B class – not great, but so far we've picked W1, W2 and B parameters quite randomly.

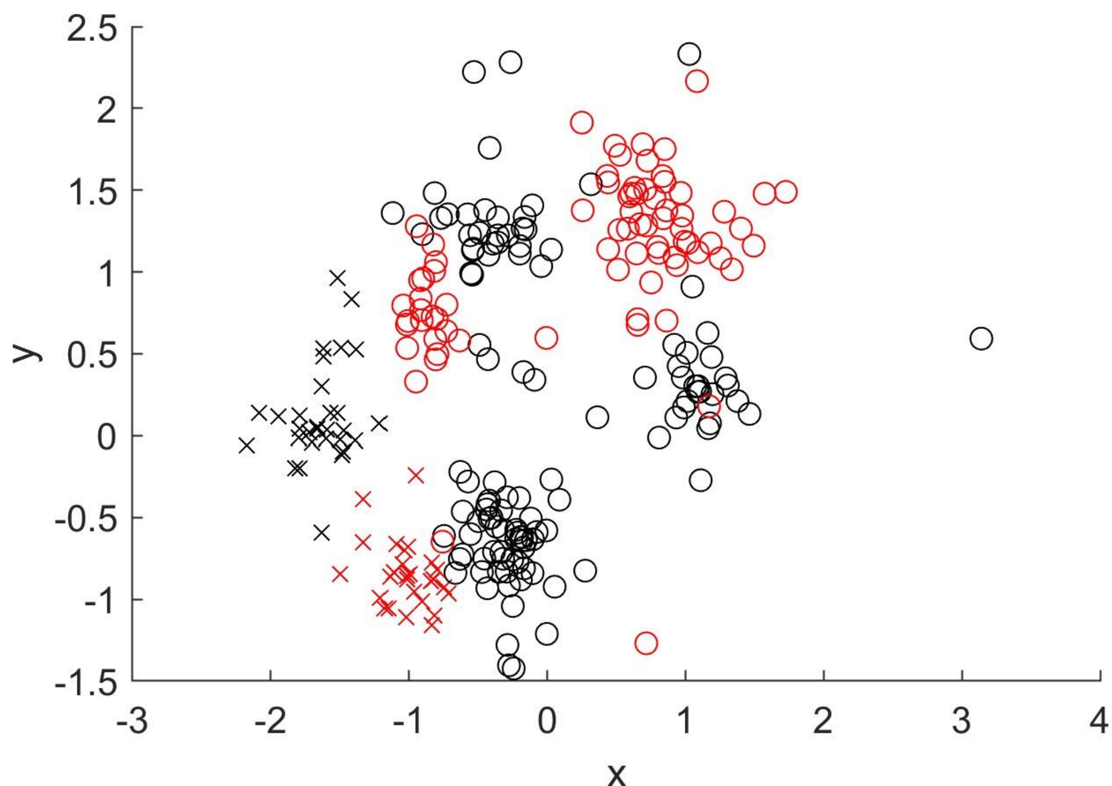

Figure 2 – Distribution of errors in our linear two-class classification problem.

Could we score better? Lets note that we have a three-parameter, one-criterion optimization problem. Our objective function that we would want to minimize is a sum of errors in both classes.

We already have tools that can deal with this kind of problems, namely: various optimization algorithms developed in scope of 1st laboratory.

In order to use any of the solutions developed before (e.g. 1+1 or grid search) we'll need to save our script for using a classifier as a function – taking parameter values as input (marked in green in the code), returning a sum of errors (marked in cyan) and commenting out lines for plotting (marked in grey). If you do it properly, the optimization algorithm should be able to draw a clear one-line division minimizing errors. Also, we will need to change VA_DATA to TR_DATA, because we will be training our algorithm now, not checking its performance. For clusters_10 the resulting image should look like in Figure 3.

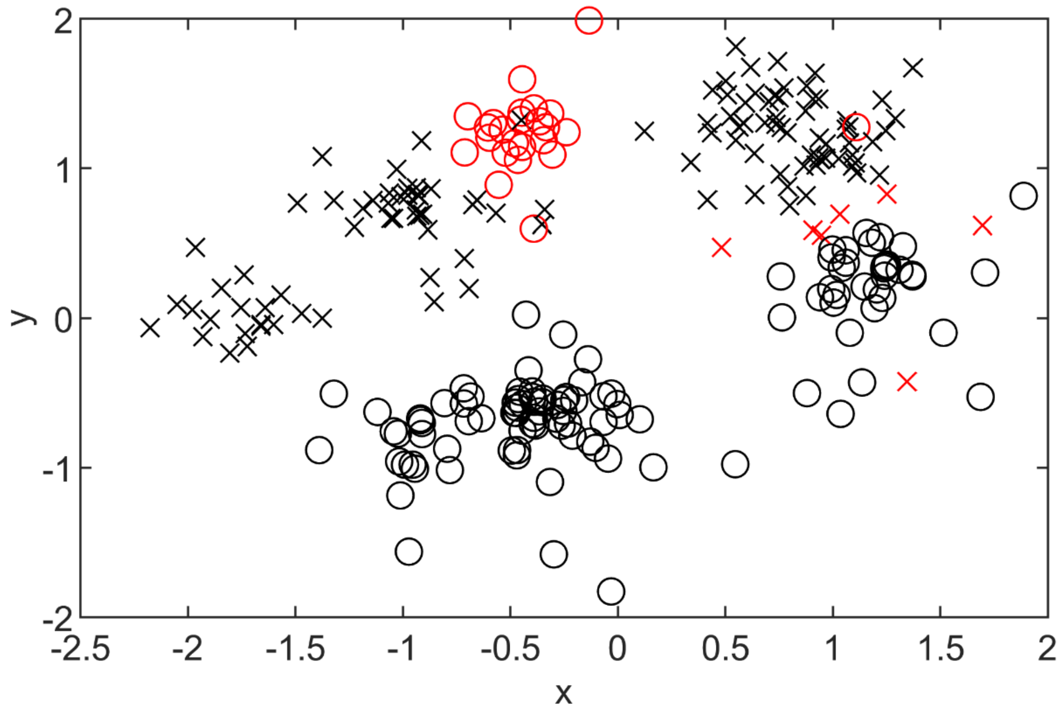

Figure 3 – Results of trained one-line classification for Clusters_10 data structure. We can see majority of errors in one spot – for one non-linearly-separable data cluster.

Task 3.2: Using script developed in laboratory 1 optimize parameters of your linear classifier. Use a **grid search** algorithm. To optimize parameters use training data (**TR_DATA**). After successful optimization test your classifier using validation data (**VA_DATA**). Save the script so it could be checked by the teacher later and store the result of this optimization in Table 3.1 at the end of this instruction.

Note that you have three parameters to optimize so your GridSearch script will require an additional loop for a 3rd parameter modification.

Task 3.3: Using script developed in laboratory 1 optimize parameters of your linear classifier. Use a **1+1** algorithm with adaptive step. To optimize parameters use training data (**TR_DATA**). After successful optimization test your classifier using validation data (**VA_DATA**). Configure optimization algorithm in such a way that it consistently finds global minimum. Save the script so it could be checked by the teacher later and store the result of this optimization in Table 3.1 at the end of this instruction.

Classifier structure allowing nonlinear classification

Remember that our clusters were not separable linearly? Now we'll find a way to actually separate all of them.

The idea that we will use is to actually do multiple classifications using many lines – and then somehow use this sub-results to derive final verdict about class presence. Look for instance in Figure 4. We could say that the class is 'blue circle' if the point is either below red line OR above orange one and at the same time below green one. Such condition would allow us to significantly improve classification accuracy – rendering only a bunch of points misclassified due to clusters overlap and outlier presence.

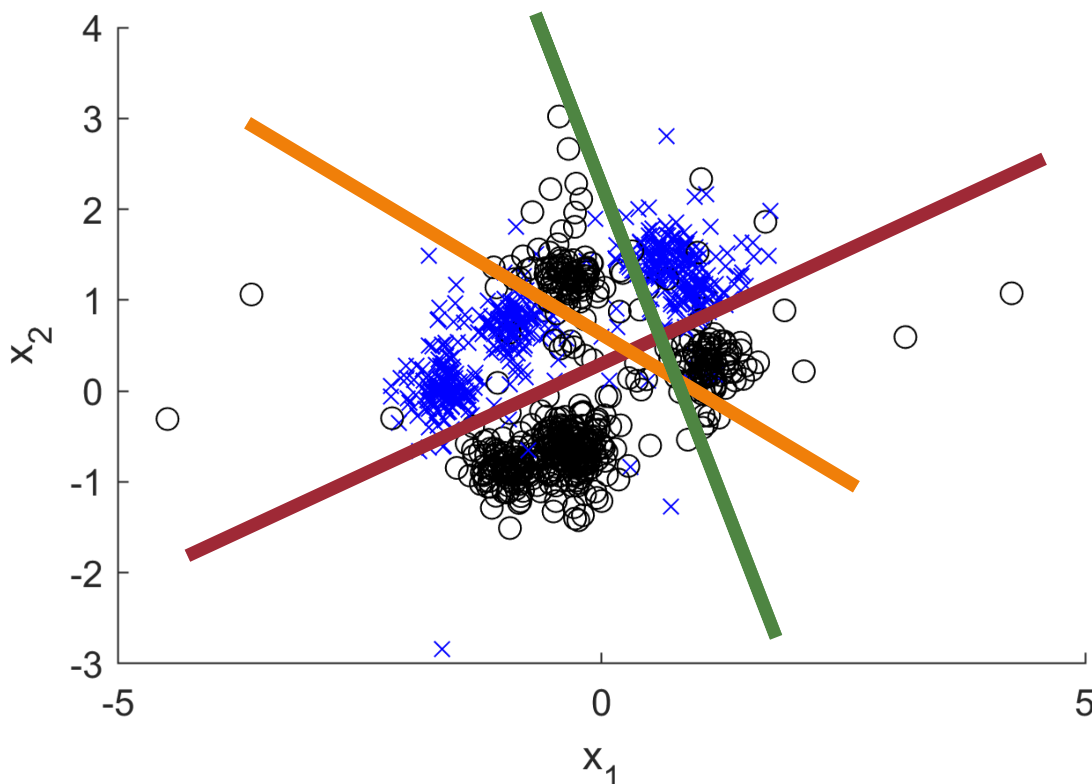

Figure 4 – Adding two additional lines enables correct classification of these samples.

Before we'll build a classifier, we first need to decide how our model would operate. We will be using *Parameters* structure which will store coefficients for all the lines. Consecutive coefficients can be coded by consecutive elements of vectors of our structure, like in the following code. Coefficients defining the same lines are given in the same colour:

```
Parameters.W1 = [ 0.2, 0.3,-0.5];
Parameters.W2 = [ 1.3, 1.1, 2.1];
Parameters.B =  [ 3.0, 0.1,-0.7];
```

We will use our previous *InitialClassifier* and modify its internal code to go through all the lines in our setup using the following instruction:

```
ClassPresence = 0;
 for k = 1:length(Parameters.W1)
    if(Parameters.W1(k)*x + Parameters.W2(k)*y + Parameters.B(k) > 0)
         ClassPresence = ClassPresence + 1;
    else
         ClassPresence = ClassPresence - 1;
    end
 end
```

If we start from *ClassPresence = 0* and after going through all the lines we are at 0 or higher – the classifier should return *ClassLabel = 1*. Otherwise *ClassLabel* should be 0. Note the *k* variable – it tracks the number of the line we are currently investigating. We will call this modified function a *MultilineClassifier* Now the only thing we need to do is to apply an optimization solution to actually train our classifier. We will need a vector of parameters equal in length to all the optimized variables (three per line). So probably we'll introduce it to our optimizer using something like this:

```
linesUsed = 9;
dimensions = linesUsed * 3;
Range = zeros(dimensions,2)
% Here we state what range we want to draw initial solutions from:
Range = Range + [-10,10]

% And a starting point:
Point = Range(:,1)' + rand(1,dimensions).*(Range(:,2)-Range(:,1))';
```

Note that since we decided to use a structure-based representation for our classifier (to make it easier to understand), we need to build it from our optimized vector representation. We can do that simply by using the following code:

```
Parameters.W1 = OptimizerOutput(1:linesUsed);
Parameters.W2 = OptimizerOutput(linesUsed +1:2* linesUsed);
Parameters.B  = OptimizerOutput(2*linesUsed +1:end);
```

If, at some point you'd like to do the reverse, you can simply use this code:

```
OptimizerInput = [Parameters.W1,Parameters.W2,Parameters.B]
```

If you are curious where the optimized lines are actually placed, feel free to use this piece of code after you plot classified points. This will draw the lines in feature space according to contents of *Parameters* structure:

```
X = [-10,10]
Y(:,1) = -(Parameters.W1(:)*X(1) + Parameters.B(:))./Parameters.W2(:);
Y(:,2) = -(Parameters.W1(:)*X(2) + Parameters.B(:))./Parameters.W2(:);
for(p = 1:linesUsed)    line([X],[Y(p,:)]); hold on;  end
```

Task 3.4: Using your **individual*** dataset for classification:

(a) Implement training of a 5-line classifier using 1+1 training method.
(b) Configure metaparameters of 1+1 solution so it would be reasonably consistent.
(c) Evaluate your final solution statistically (At least 20 tries) and store the results in structure generated in task 2.11.
(e) Save statistical estimates of your tries in Table 3.1 at the end of the instruction.

Introduction of a gradient to a classification problem

As you probably noticed, we were not using gradient algorithm for optimization. Lets find out why. Just for the sake of curiosity, lets go back to our 1-line classifier used in task 3.3 and lets pass our *ObjectiveFunction* to our gradient algorithm and see what happens (Figure 5):

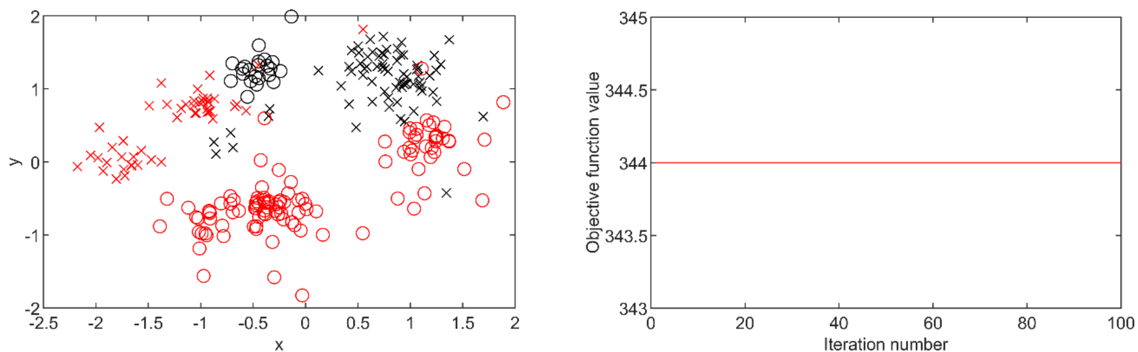

Figure 5 – Vanilla gradient used for classifier training

Apparently, the method in its current state cannot properly train our classifier... Why does it work like that?

It is worth noting that despite having a *continuous* problem (we can assign floating point values to ***W1***, ***W2*** and ***B***) we cannot use gradient descent algorithm yet. The reason for this is as follows: If we move our separation line and cross with it location of any data point the objective function value would not change gradually – instead it will be incremented or decremented. Very small change of any parameter would likely not cause any change of the objective function value. We could let it be and just refrain from using gradient-based solutions, but let's treat it as challenge and think what can we do to enable gradient approach here. For starters, we need a sub-function that will tell us not only if our classification line 'crossed' a data point, but also "how far away" the line is from the point at all times. This function we will call *activation function* and we can define it like this:

$$\boldsymbol{u} = \frac{\mathbf{2}}{\mathbf{1} + \boldsymbol{e}^{-\boldsymbol{y}}} - \mathbf{1}$$

where:

$$\boldsymbol{y} = \boldsymbol{w}_{\mathbf{1}}\boldsymbol{x}_{\mathbf{1}} + \boldsymbol{w}_{\mathbf{2}}\boldsymbol{x}_{\mathbf{2}} + \boldsymbol{b}$$

So lets code it down:

```
function[AF] = ActivationFunction(x,y,Parameters)
   AF = 2/(1+exp(-(Parameters.W1*x + Parameters.W2*y + Parameters.B))) - 1;
end
```

Now we can just plug this equation to where we used to have a check of whether the point is above or below line. Now we won't have ones or zeros here – we'll have a continuous result depending on how far away the point is from the line – with the answer approaching -1 or +1 for points placed far away. Our goal will be the same as before: To minimize objective function (*ClassificationLossCumulated)*, but this time we will not actually count errors, but results of passing distance between each data point and a classification line through our 'activation function'. Every 'good' classification should contribute negative values to the sum, every 'bad' classification will contribute positive values to the sum. Note that we can do that easily by just changing the highlighted sign (we want all the samples having activation function equal to "1" for "A" class and -1 for "B" class):

```
function[LossCumulated] = ClassificationLossCumulated(Input)

    load TR_DATA
    Parameters.W1 = Input(1);
    Parameters.W2 = Input(2);
    Parameters.B =  Input(3);
    LC = 0;

    for k = 1:length(TR_DATA)
     if(TR_DATA(1,k) == 1)     % For class A
       LC = LC - ActivationFunction(TR_DATA(2,k),TR_DATA(3,k),Parameters);
     else   % For class B
       LC = LC + ActivationFunction(TR_DATA(2,k),TR_DATA(3,k),Parameters);
     end
    end
   LossCumulated = LC;
end
```

If we do that, we observe that our classifier actually learns the underlying pattern. While it can sometimes end up in a local minimum, the convergence curve will show improvement for at least several of the initial iterations (see Figure 6).

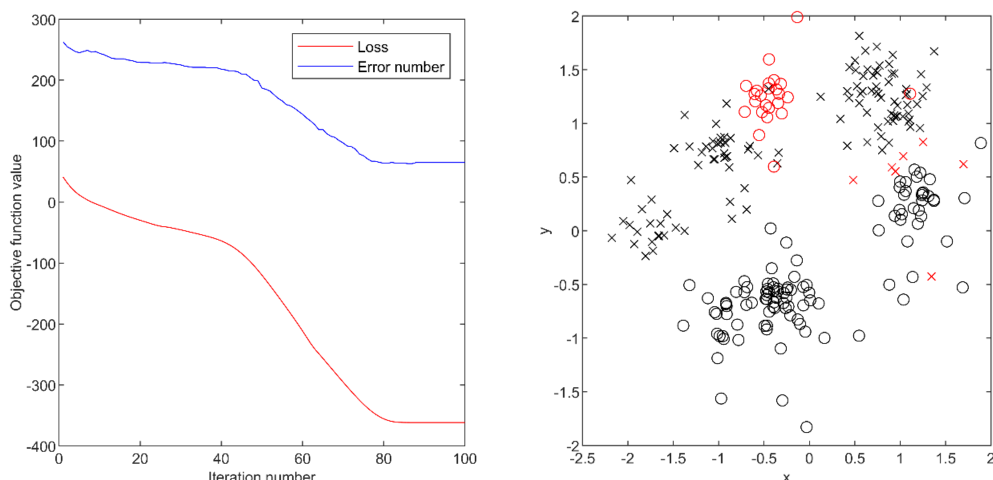

Figure 6 – Linear classifier fitted using an activation-function-based gradient descent method

Task 3.5: Add the sigmoid function to enable gradient-descent optimization usage in our classifier. Now configure and test your new gradient-based classification algorithm and store the results in Table 3.1 (Three tries of the method)

Now, lets generalize our solution so it would allow for a multiline classification. Again, we need just a few small adjustments. We will be using the gradient descent code from 1st instruction with changes similar to ones necessary in task 3.5. What will be different, is the classifier and the actual function calculating quality of our solution. To evaluate if point should be classified to one or the other class, we need the following classifier:

```
function [CumulativeActivation] = MultilineClassifierDifferentiable(x,y,Parameters)
   CumulativeActivation = 0;

   for k = 1:length(Parameters.W1)
      AF = 2/(1+exp(-(Parameters.W1(k)*x + Parameters.W2(k)*y + Parameters.B(k)))) - 1;
      CumulativeActivation = CumulativeActivation + AF;
   end
    % In order to cast the solution to the (0,1) range maintaining differentiability:
   CumulativeActivation = 1/(1+exp(-(CumulativeActivation)));
end
```

This basically works the same as a M*ultilineClassifier* – we just have a sigmoidal function in a (-1,1) range (marked in cyan) instead of checking if point is above or below the line And then, since we expect the classifier to return values from a (0,1) range (indicating if its class 0 or 1), we need to cast the result on this range using the yellow sigmoid.

Now, the actual training function that goes through all the points from a training range can be very simple:

```
function[Loss] = TrainTheClassifierGradient(Parameters)

load TR_DATA
Loss = 0;
     for k = 1:length(TR_DATA)
        if(TR_DATA(1,k) == 1)     % For class A
          Loss = Loss + (1 - MultilineClassifierDifferentiable(TR_DATA(2,k),TR_DATA(3,k),Parameters));
        else  % For class B
          Loss = Loss + MultilineClassifierDifferentiable(TR_DATA(2,k),TR_DATA(3,k),Parameters);
        end
    end
end
```

Notice that *Loss* variable just counts errors in such a way, that it cumulatively adds results for class B (because they should be 0) and adds differences from 1 for class A (because they should be 1).

Task 3.6: Repeat task 3.4 for your **individual** dataset, this time using a gradient-descent method.

(a) Implement training of a 5-line classifier using gradient descent method.
(b) Configure metaparameters of gradient descent so it would be consistent.
(c) Evaluate your final solution statistically (At least 20 tries) and store the results in structure generated in previous laboratory.
(e) Save statistical estimates of your tries in Table 3.1 at the end of the instruction.

Table 3.1: Linear classification

	W1	W2	B		
Grid search				**Sum of errors:**	
1+1				**Sum of errors:**	
Multiline 1+1	-				
Gradient (1st try)					
Gradient (2nd try)					
Gradient (3rd try)					
Multiline gradient	-			**Mean:**	
				Std:	

Additional tasks:

What happens to training difficulty if we increase dimensionality of our problem?

OK, we have a working solution for non-linear classification. The question which feels important, however, is: what will happen if we will need our classifier to use much more lines? Will it make the problem harder? If yes – how much harder? Lets test it and see what happens. We will increase number of lines in our classifier and compare repeatability of solutions based on 1+1, gradient and multistart gradient approaches. To keep things a bit simpler, lets force ourselves to work with a particular number of starts in a multistart method (lets say: 5) and to be fair, we'll give the same number of objective function checks to all our algorithms. Lets now test their performance and see what happens…

Task 3.7: Evaluate statistically a 5, 9, 13, 17, 21 and 25 line classifiers using 1+1, gradient and multistart gradient algorithms in the best configuration you can provide. Save the results to our data structure (including statistical estimates). Draw conclusions and explain what is the insight we get from this experiment.

Is the multiline problem difficult because of local minima or exploitation difficulty?

We should probably investigate this multidimensional optimization problem a bit deeper. For starters, choice of number of starts of a multistart method is not that obvious in high-dimensional spaces. While intuitively it might feel like the more dimensions, the harder the problem gets, sometimes it is not the case…

Task 3.8: Configure a multistart gradient training algorithm that has 1500 objective function checks for optimization of a 25-line classification problem. Pick number of starts equal to 1, 3, 5, 10, 15 and 30 and adjust remaining parameters so it would provide results as consistent and good as possible. Then do the same for a 5-line classification problem. Draw conclusions and explain what is the insight we get from this experiment.

Adding momentum to our optimization

The basic gradient algorithm method is not utilizing momentum yet. Lets add it to our code. We want to add to every step the weighted value of a previous step according to the following formula:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla f(\theta_t)$$

Where:
v_t is the velocity of parameter change (that is: the actual step)
β is the weight of momentum
$\nabla f(\theta_t)$ is the gradient of the objective function at parameters θ_t

Now, v_t can be added to our current coordinates using our learning rate:

$$\theta_{t+1} = \theta_t - \eta v_t$$

Probably a good idea would be to start again from a simple 2D optimization function so we'd know if the method is working as expected. Once we have the working solution for a 2D optimization problem, we can implement it easily also for our multidimensional classifier-training problem.

How should we choose weight of the momentum? We of course need to have this value in range of 0 (no momentum influence) to almost 1 (only momentum influence). A good starting (default) point would be $\beta = 0.9$. Note that our learning rate can start much lower now – because our algorithm will speed-up by itself.

Task 3.9: Add momentum to your gradient algorithm, evaluate it statistically in task of optimizing a multiline classifiers and add its results to the data structure and to the table 3.1