Wydział Inżynierii Mechanicznej i Robotyki
Katedra Robotyki i Mechatroniki

Signal processing and identification in control of mechatronic devices

Signal processing and identification in monitoring of mechatronic devices

Temat: Soft computing tools in classification of vision data

Aim: Feature selection and classification of objects in a prepared feature space

Issues covered: Feature space visualization, feature selection, letter classification, decision tree classifier, kNN classifier, Multilayer perceptron, optimization of metaparameters of methods, classifier testing, overfitting, transfer learning

LA: dr inż. Ziemowit Dworakowski

Introduction

For the purpose of the exercise we'll use letter images. The database consists of the following letters grouped into three groups:

Group 1: Letters F H K L M N T and S
Group 2: Letters A and D
Group 3: Letters E G and R (not contained in a students' version of the dataset)

From among letters of 1st group the **individual** classification task are built. In order to determine the task to do one should divide number of letters of his or her name and surname by 8. The remainders of these divisions will point to classes that needs to be distinguished, so 1 = F, 2 = H and so on up until 0 = S.

For instance *Jane Doe* will have a classification task consisting of letters **L** and **K** (Remainder of division 4 (*"Jane"*) by 8 is 4, 4th letter in group 1 is **L**. Remainder of dividing 3 (*"Doe"*) by 8 is 3, 3rd letter of group 1 is **K**. In case when one would have the same number of letters in his or her name and surname, second class is obtained by adding 1 to the remainder, for instance *Roger Moore* would have letters **M** (remainder is 5) and **N** (Remainder is 5 + 1).

All of the data are in structures named accordingly, so data for letter **A** (letter image and pre-extracted features) are in structure **FeaturesA**. The set contains letters written in different fonts, in bold, italics or both or written in capitals. In addition to that, there are three subsets. A *Normal* subset containing base data and subsets *W1* and *W2* containing distorted versions of the letters.

Pre-extracted features include:

Basic *regionprops* features:

Area, MajorAxisLength, MinorAxisLength, Eccentricity, Orientation, ConvexArea, Circularity, EulerNumber, EquivDiameter, Solidity, Extent oraz Perimeter

Image of the letter (it can be used in order to extract further more advanced features or for deep learning)

Image

Moment-based features and moment invariants:

Moments: A structure with central geometrical moments (M...) and normalized central moments (N...)
HuInvariants: A structure with 6 initial Hu Invariants (I...)

Visualization of features in 2D spaces

In order to explain how to approach **individual** tasks, we'll use an exemplary task of classification A and D letters. The initial step would be to familiarize with the data. To do that we'll first load the data:

```
A = load('StudentData/FeaturesA');
D = load('StudentData/FeaturesD');
```

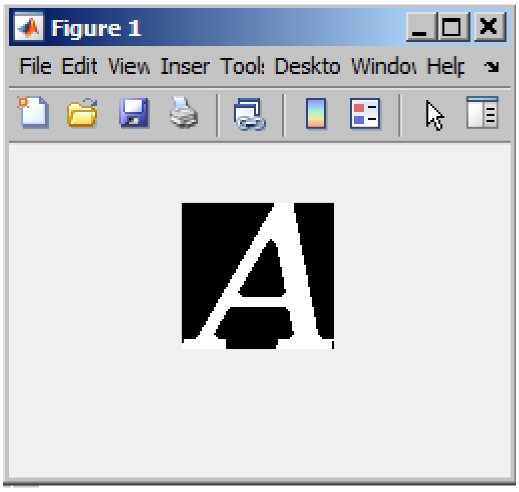
Now, in a workspace we have two structures. In order to access them (e.g. to see the particular letter instance) we could use the following code:

```
ObjectNumber = 1;
imshow(A.Data(ObjectNumber).Normal.Image)
```

Using that we've displayed the initial object of the "A" class. Distorted images of this letter are in here, but we won't use this for now:

```
imshow(A.Data(ObjectNumber).W1.Image)
imshow(A.Data(ObjectNumber).W2.Image)
```

The displayed image from a "Normal" class should look like this:



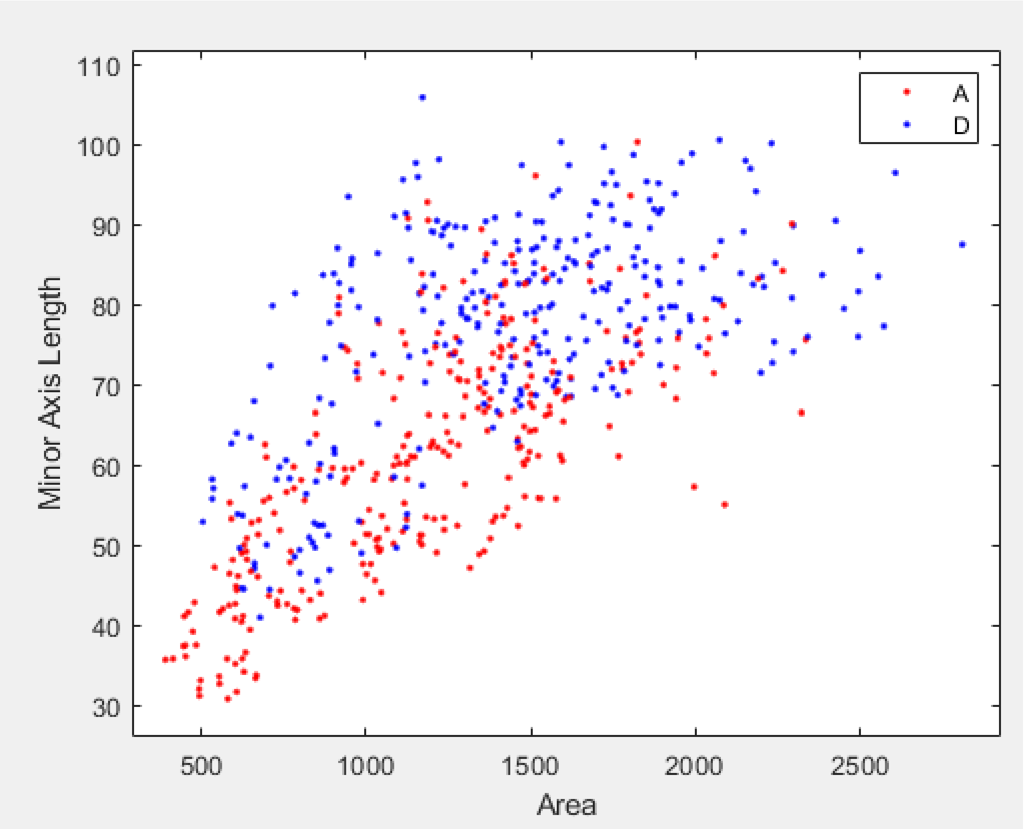
Task 1: Look into your data (data for your **individual** classification task). By modifying *ObjectNumber* to several randomly picked numbers try to identify some of letter examples that you perceive as "difficult" or "easy" for the classifier. Prepare to show these examples to the LA (in the form of sets of printscreens or subplot containing all of the chosen data). Explain to the LA your reasoning – why do you think particular examples are difficult/easy.

Because the set contain as far as 600 objects in each class, detailed investigation of all the examples is not possible. For that reason, to view the set as a whole we need to look into feature space and from now on look only for patterns visible in the selected dimensions of the feature space.. Using the following code we can visualize area and orientation of the initial 300 objects:

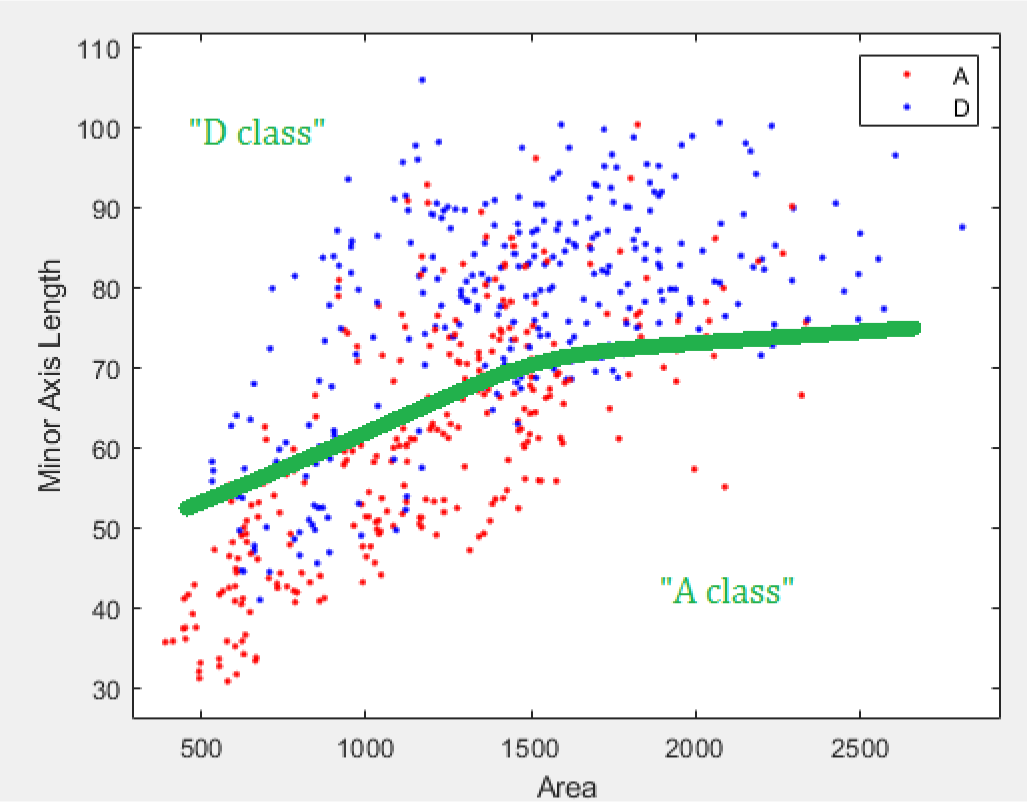
```
figure;
for k = 1:300
   plot(A.Data(k).Normal.Area,A.Data(k).Normal.Orientation,('.r')); hold on
   plot(D.Data(k).Normal.Area,D.Data(k).Normal.Orientation,('.b')); hold on
end
xlabel('Area');
ylabel('Orientation');
legend('A','D');
```

Note! We see only the initial 300 objects, not all of them. We do it on purpose. Why – we'll see in a few moments. Right now please follow this guideline and **DO NOT** change the “300” in the code to “600”!

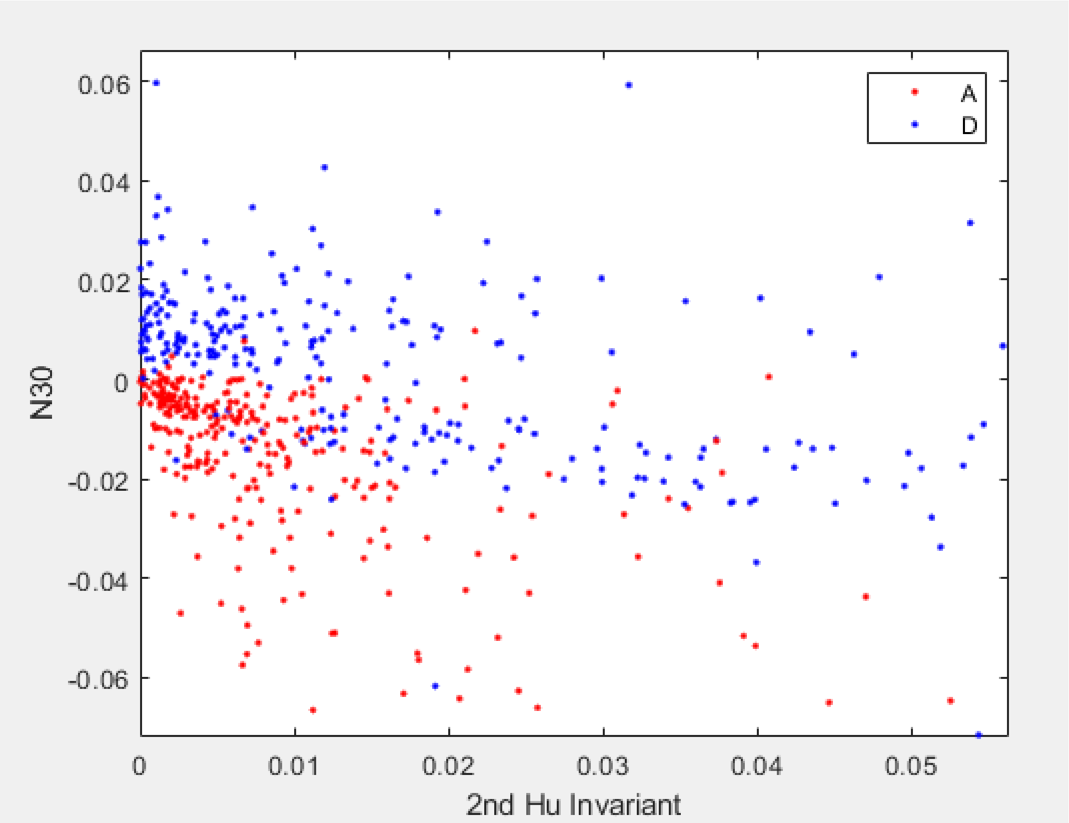
The obtained result should look as in here:



We can see that the “A” and “D” objects do not occupy the same area – the “D” cluster appears to be moved a bit higher and to the right. Maybe we could classify the objects in this space and the obtained classification efficiency would be higher than random (higher than 50%):



But the aim of this stage is to find such features that allow for the easiest division possible. After a few tries and testing few features from *Moments* group it was possible to find the following set of features:



In here the results are much closer to the desired – there are only a few areas in which the classes overlap. We of course aim for a possibility of a linear classification with 0 errors, so cluster for “A” and “D” far away from each other, but in vast majority of cases this will not be possible to obtain. For now we'll settle on this feature set and we'll see what we can do with it later.

Task 2: Visualize different 2-dimensional feature spaces for initial 300 objects of both classes of your **individual** classification task. Identify the spaces in which the classes are separated as good as possible. Save plots of 2-3 examples that you deem the best. Consult the LA for choice of the best feature set to proceed.

Note! Do not try to copy features used by your colleagues and do not try to obtained a similar-looking result. Different tasks are to be solved with different features. The final efficiency will also vary significantly. In some tasks almost-perfect division is possible. In others clear separation of clusters will not be possible. You are graded by the correctness of steps taken, not by the obtained efficiency of classification.

Decision tree classfier:

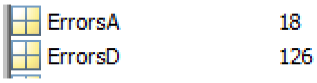
Based on the selected feature subset we'll propose classification rules that will allow for class division. In our example we can use the fact that central normalized moment N30 appears to be positive values for “D” and negative for “A”. Lets build a simple classifier and save it as a separate function:

```
function[Class] = ClassifierDT(Object)
    if(Object.Moments.N30 > 0)
        Class = 1;
    else
        Class = 0;
    end
end
```

Note, that this classifier is built similarily to the classifier from 3rd instruction, with the exception of parameters. Then, we had parameters defining classification rules (W1, W2 and B) passed as a function argument. Now, the classification rules are hard-coded in the classifier. Because of that, its autonomous training will not be possible, but we will be able to easily adjust its parameters manually – by changing the classification condition inside a function. We also got rid of a 3-parameter definition of a line in favor of simple classification perpendicular to particular axis. This again makes training challenging (we strip the classifier of “degrees of freedom” thus make it much less flexible) but we allow for easy and intuitive understaning of what each condition mean. Now we can feed it with data and see how many mistakes will it make:

```
ErrorsA = 0;
ErrorsD = 0;
for k = 1:300
    if(ClassifierDT(A.Data(k).Normal) == 1)   % Detected class "1", (our "D")
        ErrorsA = ErrorsA + 1;
    end
    if(ClassifierDT(D.Data(k).Normal) == 0)   % Detected class "0", (our "A")
        ErrorsD = ErrorsD + 1;
    end
end
```

In our case we have the following numbers of errors:



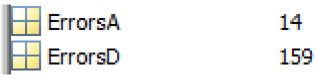
```
ErrorsA          18
ErrorsD          126
```

So class A is recognized efficiently but in D class we have rougly around 42% of errors. After summing up we have 76% efficiency (144 errors, 600 samples in total).

Up to this moment we've worked on half the data at most. This was our training dataset based on which we adjusted parameters of our classifier. Lets see how will our classifier work on data unseen up until now. In order to do that lets classify data from the second part of dataset:

```
ErrorsA = 0;
ErrorsD = 0;
for k = 301:600
    if(ClassifierDT(A.Data(k).Normal) == 1)  % Detected class "1", (our "D")
        ErrorsA = ErrorsA + 1;
    end
    if(ClassifierDT(D.Data(k).Normal) == 0)  % Detected class "0", (our "A")
        ErrorsD = ErrorsD + 1;
    end
end
```

That rendered the following result:



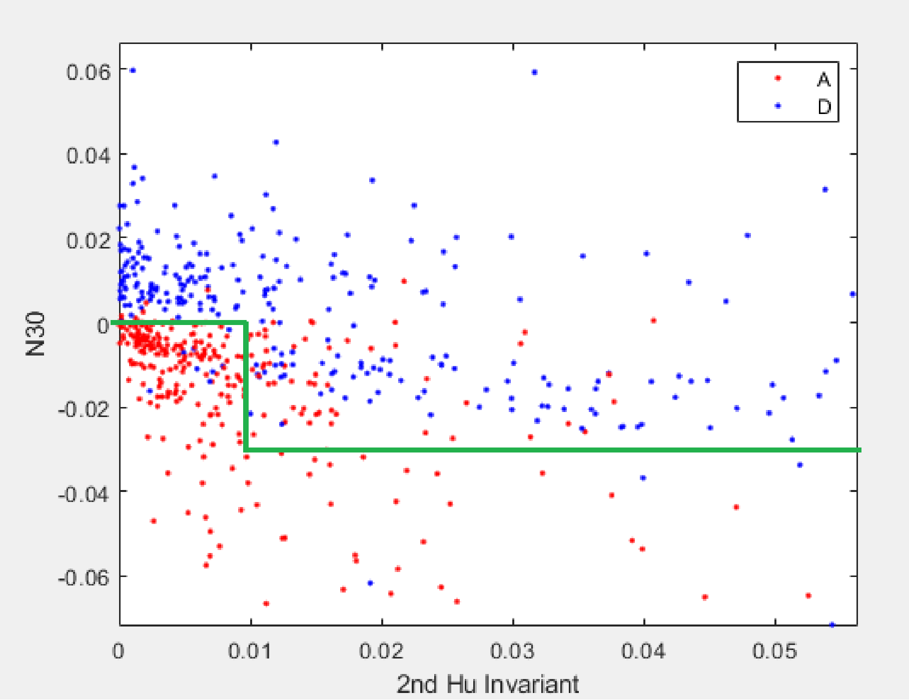
```
ErrorsA    14
ErrorsD    159
```

We thus can see that the solution is general (the level of errors is similar to one obtained in our manual “training”)

Of course this code is only an idea demonstration. In the classifier we could use more than one feature and combination of several classification conditions. For instance classifier build like this:

```
function[Class] = ClassifierDT2(Object)
    if(Object.Moments.N30 > 0)
        Class = 1;
    elseif(Object.Moments.N30 < -0.03)
        Class = 0;
    elseif(Object.HuInvariants.I1 < 0.01)
        Class = 0;
    else
        Class = 1;
    end
end
```

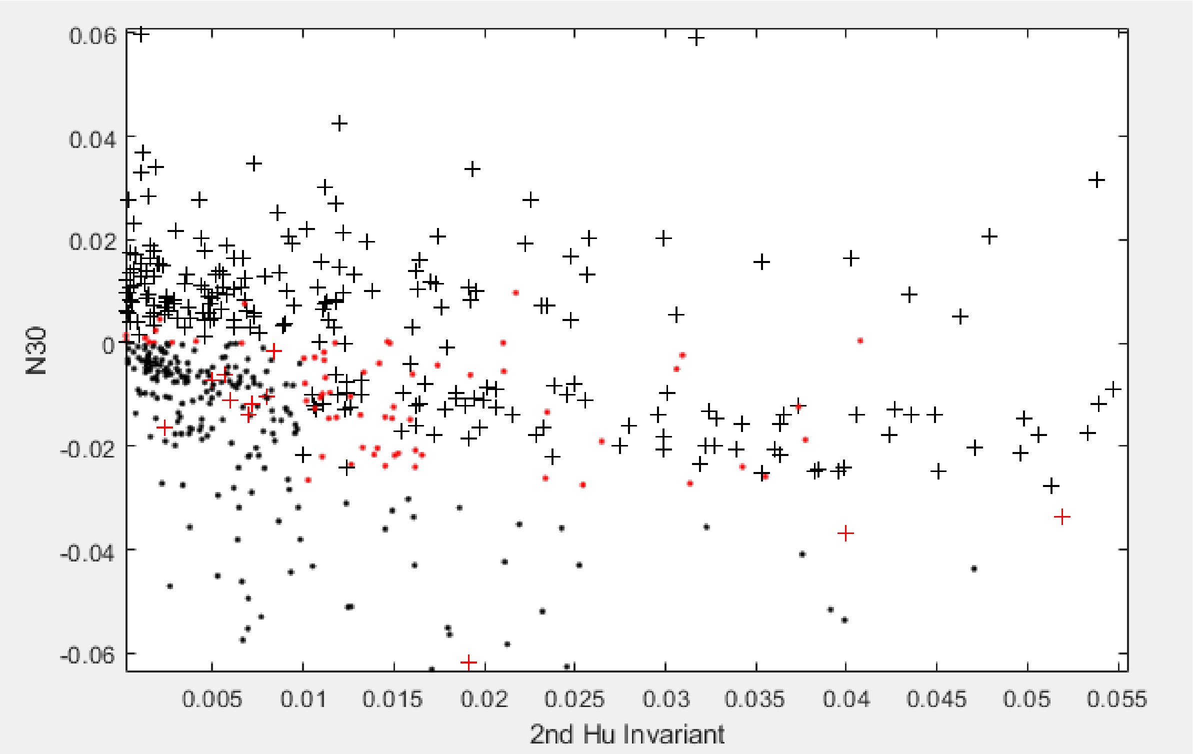
allows for classification of space acording to this condition:



As a result we've obtained only 93 errors in training data and 115 in testing. In order to visualization of classification and highlight the areas in which the errors are located we can use the following (highlighted) addition to code:

```
ErrorsA = 0;
ErrorsD = 0;
figure;
for k = 1:300
    if(ClassifierDT2(A.Data(k).Normal) == 1)  % Detected class "1", (our "D")
        ErrorsA = ErrorsA + 1;
        plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.r'); hold on
    else
        plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.k'); hold on
    end
    if(ClassifierDT2(D.Data(k).Normal) == 0)  % Detected class "0", (our "A")
        ErrorsD = ErrorsD + 1;
        plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+r'); hold on
    else
        plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+k'); hold on
    end
end
xlabel('2nd Hu Invariant');
ylabel('N30');
```

which allow for checking error locations:



Now we could try to move the conditions in our classifier so the number of errors in training dataset would be minimal. After quick adjustments number of errors for training / testing data fell down to 77 and 99, respectively, for the following classifier:

```
function[Class] = ClassifierDT(Object)
    if(Object.Moments.N30 > 0);                    Class = 1;
    elseif(Object.Moments.N30 < -0.035);           Class = 0;
    elseif(Object.HuInvariants.I1 < 0.016);        Class = 0;
    else                                           Class = 1;
    end
end
```

Task 3: Based on your two selected features (for **individual** task) propose a classifier based on a decision tree. Configure parameters based on 300 initial examples of each class. Try to obtain as low number of errors in this dataset as possible. Save the initial configuration of the classifier and the final configuration that maximizes the performance based on the training data. Then test both classifiers using a second half of dataset. Save the obtained efficiencies.

K-nearest neighbors classifer

In order to classify our objects we could use the principle of distance of object in feature space to its neighbors. We'll implement a simple k-nearest classifier to this end. We'll build a formal training dataset from our data. Note, that we again use only half of our data here:

```
for sample = 1:300
   TrainingDataClass0(:,sample) = ...
[A.Data(sample).Normal.Moments.N30,A.Data(sample).Normal.HuInvariants.I1];
   TrainingDataClass1(:,sample) = ...
[D.Data(sample).Normal.Moments.N30,D.Data(sample).Normal.HuInvariants.I1];
end
```

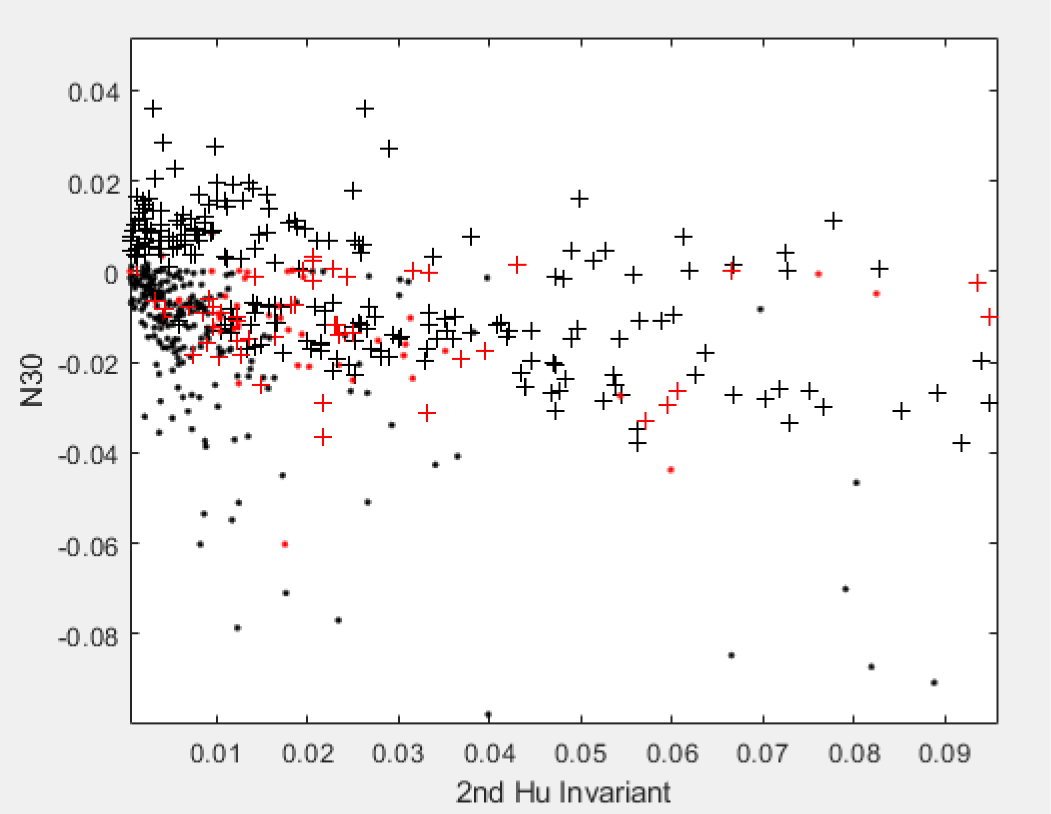
Next, we can implement a classifier that for now will look for just one closest neighbor in training data. Remember, to save it as a separate function:

```
function[Class] = ClassifierKNN(Features, TrainingDataClass0,TrainingDataClass1)
    distancesC1 = dist(Features,TrainingDataClass0);
    distancesC2 = dist(Features,TrainingDataClass1);
    if(min(distancesC1) < min(distancesC2))
        Class = 0;
    else
        Class = 1;
    end
end
```

Finally, we can test it in our task:

```
figure;
for k = 301:600
    F1 = A.Data(k).Normal.Moments.N30;
    F2 = A.Data(k).Normal.HuInvariants.I1;

    if(ClassifierKNN([F1,F2],TrainingDataClass0,TrainingDataClass1) == 1)  % Detected class "1"
       ErrorsA = ErrorsA + 1;
       plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.r'); hold on
    else
       plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.k'); hold on
    end

    F1 = D.Data(k).Normal.Moments.N30;
    F2 = D.Data(k).Normal.HuInvariants.I1;

    if(ClassifierKNN([F1,F2],TrainingDataClass0,TrainingDataClass1) == 0)  % Detected class "0
       ErrorsD = ErrorsD + 1;
       plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+r'); hold on
    else
       plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+k'); hold on
    end
end
xlabel('2nd Hu Invariant');
ylabel('N30');
```

Note, that classifier build in such a way will always score 100% efficiency in our training data. In our testing data, however, we've obtained efficiency of 82% (104 errors):



Task 4: Based on your 2-feature space implement and test kNN classifier (with k = 1). Now add more features to your classifier: let the classifier work in 4-dimensional feature space, 2 initial features being the features that you've used up until now and 2 added features taken from alternative good features found in scope of task 1. Does it allow for increase of efficiency? Store the codes of both classifiers in order to show the results and codes to the LA

Note! Our classifier is already prepared to work in feature space with any number of dimensions! We don't need to adjust anything in the classifier code, we only need to add more features in new columns of *TrainingDataClass0* and *TrainingDataClass1* matrices and add more features (e.g. like this: `[F1,F2,F3,F4]`) in calling of our classifier.

When we use distance-based classifiers usually a **normalization** of data is crucial. In our case majority of data falls into similar ranges on both X and Y axes (roughly around 0.04 units wide in our example) but it is possible that these ranges will be significantly different. In such a case a feature that has much higher spread will contribute to the distance to the greater extent changing effectively our classification space into just one-dimensional. The simplest aproach that solves this problem is to normalize data by dividing each feature value by the maximum value in the used dataset (or mean value – which is less sensitive to outliers). Of course when using this approach, we'll have to consistently divide samples in testing dataset by the same value as well – so if we calculate a mean to divide using a training dataset, we need to use the same already-calculated value for testing data, and not calculate it again for testing dataset.!

Normalization procedure for our data could be done using the following code:

```
% Finding of the means for the whole dataset
Means = abs(mean([TrainingDataClass0,TrainingDataClass1]'));
% Dividing the data:
TrainingDataClass0 = TrainingDataClass0./Means'
TrainingDataClass1 = TrainingDataClass1./Means'
```

Next, feature extraction can look as in here:

```
    F1 = A.Data(k).Normal.Moments.N30/Means(1);
    F2 = A.Data(k).Normal.HuInvariants.I1/Means(2);
```

Task 5: Normalize your data. Does that affect the classifier efficiency? Store the code so you'll be able to present the results to the LA.

As its name suggests, the “kNN” classifier should be able to use a “k” neighors instead of just 1. In order to implement such a classifier we'll modify its code in this way:

```
function[Class] = ClassifierKNN(Features, TrainingDataClass0,TrainingDataClass1)

    K = 3;   % Metaparameter: How many neighbors do we want to take?

    distancesC1 = dist(Features,TrainingDataClass0);
    distancesC2 = dist(Features,TrainingDataClass1);
    % We store all the calculated distances into one matrix
    TestingMatrix(:,1) = [distancesC1,distancesC2];
    % We add information from which class do rows come from
    TestingMatrix(:,2) = [zeros(1,length(distancesC1)),ones(1,length(distancesC2))];
    % We sort the matrix so the closest distances are in the top
    SortedMatrix = sortrows(TestingMatrix,1)
    % We take K top rows and sum class indications
    CompoundNeighborClasses = sum(SortedMatrix(1:K,2));
    % if the sum is higher than half of the K - we have class 1
    if(CompoundNeighborClasses < K/2)
        Class = 0;
    else
        Class = 1;
    end
end
```

Task 6: Test a kNN classifier for k = 1, k = 3, k = 5 and k = 9 in two, four and six-dimensional feature space. Store the obtained results (number of errors and percent efficiency) in the following table: How well our classifier scales for higher-dimensional spaces? Does the increase in k affects the classifier in any way?

	k = 1	k = 3	k = 5	k = 9
2 features				
4 features				
6 features				

Multilayer perceptron

The next tool that we'll use is a simple neural net called a Multilayer Perceptron (MLP). It is worth to note that the code that we've prepared for testing the kNN classifier is already fit to use for MLP. We'll need a training data that we'll use from the already-prepared *TrainingDataClass0* and *TrainingDataClass1* matrices:

```
TrainingData = [TrainingDataClass0,TrainingDataClass1];
TargetTrain = [ones(1,300),zeros(1,300)];
```

The proper size of the TrainingData matrix is 2 x 600 (if we use 2 features) or 4 x 600 (if we use 4 features). The proper size of a TargetTrain vector is 1x600.

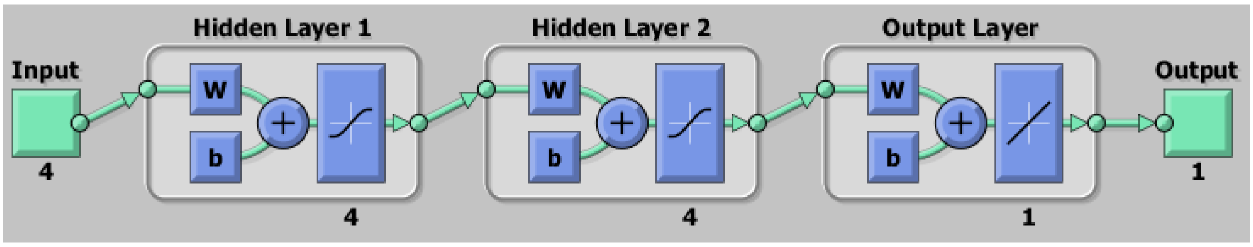
Next, we need to adjust metaparameter values for the net and configure it:

```
NetworkSize = [4 4];          % Number of neurons in all the hidden layers
OurNet = newff(TrainingData,TargetTrain,NetworkSize); % Net initialization
OurNet.TrainParam.time = 10;     % Max time allowed [in sec]
```

We can see how our net looks using the following command:

```
view(OurNet)
```

For 4-feature net we should see something that looks like this:



Other, suspiciously higher number of inputs likely means that training data is in a “vertical” form (rows contain data for consecutive letters) – which is a mistake.

Finally, we can train our net using this command:

```
OurNet = train(OurNet,TrainingData,TargetTrain);
```

And now we can go back to our code and modify it according to the highlighted areas:

```
for k = 301:600
    F1 = A.Data(k).Normal.Moments.N30/Means(1);
    F2 = A.Data(k).Normal.HuInvariants.I1/Means(2);

    if(sim(OurNet,[F1,F2]') > 0.5)  % Detected class "1", (our "D")
        ErrorsA = ErrorsA + 1;
        plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.r'); hold on
    else
        plot(A.Data(k).Normal.HuInvariants.I1,A.Data(k).Normal.Moments.N30,'.k'); hold on
    end

    F1 = D.Data(k).Normal.Moments.N30/Means(1);
    F2 = D.Data(k).Normal.HuInvariants.I1/Means(2);

    if(sim(OurNet,[F1,F2]') <= 0.5)  % Detected class "0", (our "A")
        ErrorsD = ErrorsD + 1;
        plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+r'); hold on
    else
        plot(D.Data(k).Normal.HuInvariants.I1,D.Data(k).Normal.Moments.N30,'+k'); hold on
    end
end
xlabel('2nd Hu Invariant');
ylabel('N30');
```

Task 7: Based on the chosen feature set (initial 2 best features) test MLP classifier. Run the code a few times. Is the result the same or different? Save your program so you'll be able to show it to the LA.

Now we know that MLP net needs to be trained and tested multiple times so we could assess it properly. Lets modify the previous code, so the section of code responsible for initialization, training and testing of the net was run 10 times. We'll also get rid of the visualization of results, and since we won't need to see the results any more, we'll pass the testing data to the net in matrix form so we'll speed-up the computations.

We'll start from definition of a testing matrix, in a similar fashion as we've prepared our training data. Note that testing data are under 301:600 indices hence this “300+” piece of code:

```
for sample = 1:300
   TestingDataClass0(:,sample) = ...
[A.Data(300+sample).Normal.Moments.N30,A.Data(300+sample).Normal.HuInvariants.I1];
   TestingDataClass1(:,sample) = ...
[D.Data(300+sample).Normal.Moments.N30,D.Data(300+sample).Normal.HuInvariants.I1];
end

TestingDataClass0 = TestingDataClass0./Means'
TestingDataClass1 = TestingDataClass1./Means'
```

Now we'll cycle through 10 initializations, trainings and tests of our net:

```
NetworkSize = [4 4];         % Number of neurons in all the hidden layers
for Test = 1:10
    OurNet = newff(TrainingData,TargetTrain,NetworkSize); % Net initialization
    OurNet.TrainParam.time = 20;    % Max time allowed
    OurNet = train(OurNet,TrainingData,TargetTrain);

    Results0 = sim(OurNet,TestingDataClass0);
    Results1 = sim(OurNet,TestingDataClass1);

    ErrorsA(Test) = sum([Results0>0.5]);
    ErrorsD(Test) = sum([Results1<=0.5]);
    ErrorSum(Test) = ErrorsD(Test)+ErrorsA(Test)
end
[mean(ErrorSum) std(ErrorSum)]
```

Task 8: Test your net 10-times using the initial set of parameters. Now test a bigger net (eg. *NetworkSize = [10 10]*), using more features (eg. 4), using a deeper net (eg. *NetworkSize = [4 4 4]*) or mor shallow one (eg. *NetworkSize = [4]*). In which of these tasks the mean efficiency was the highest?

Now we've finally arrived to optimization of metaparameters of the MLP. The questions arises: Which metaparameter values would be optimal for the task at hand? Remembering that just one test will not give us any information regarding the efficiency of the net, we'll have to test many possible sets of metaparameter values many times. Additionally, we'll have to modify our testing data and divide it into two subsets: A subset for optimization of metaparameters and a subset for final test of the configured net. This division can be done randomly:

```
Indices = randperm(300); % Here we randomly permute data indices
for sample = 1:150
   TestingDataClass0(:,sample) =  ...
      [A.Data(300+Indices(sample)).Normal.Moments.N30,...
      A.Data(300+ Indices(sample)).Normal.HuInvariants.I1];
   TestingDataClass1(:,sample) =  ...
      [D.Data(300+Indices(sample)).Normal.Moments.N30,...
      D.Data(300+Indices(sample)).Normal.HuInvariants.I1];
end
for sample = 1:150
   FinalTestClass0(:,sample) = ...
      [A.Data(300+Indices(sample)).Normal.Moments.N30,...
      A.Data(300+Indices(sample)).Normal.HuInvariants.I1];
   FinalTestClass1(:,sample) = ...
      [D.Data(300+Indices(sample)).Normal.Moments.N30,...
      D.Data(300+Indices(sample)).Normal.HuInvariants.I1];
end
```

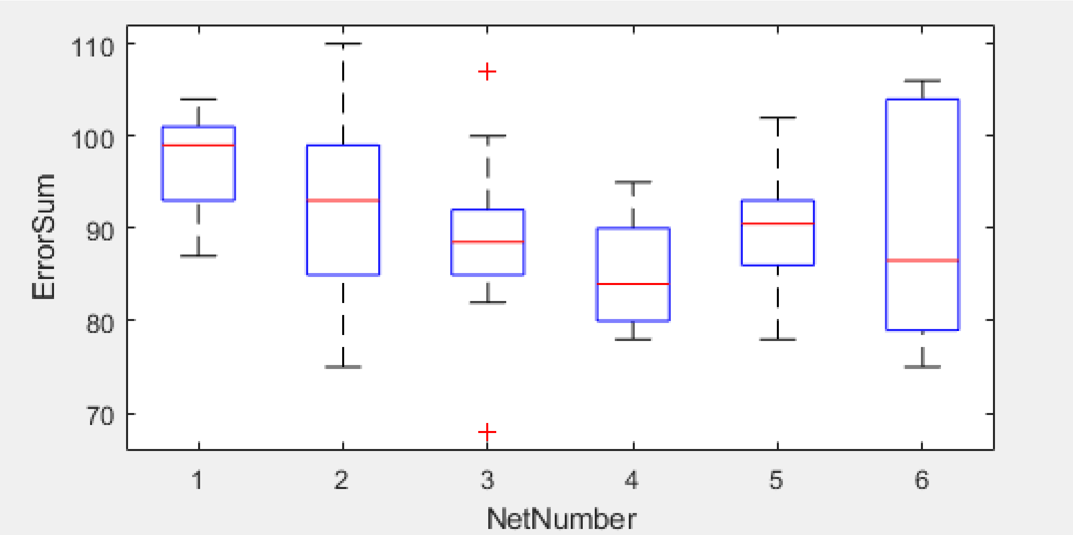
The manual optimization of metaparameters we'll do using a semi-grid-search approach modifying one metaparameter-at-a-time and seening how does it affect the result. We'll do it in a loop cycling through the metaparameter values. Lets pick 6 different sizes of the net:

```
for TestSerie = 1:6
NetworkSize = [2*TestSerie 2*TestSerie];   % Number of neurons in all the hidden layers
    for Test = 1:10
        OurNet = newff(TrainingData,TargetTrain,NetworkSize); % Net initialization
        OurNet.TrainParam.time = 20;   % Max time allowed
        OurNet = train(OurNet,TrainingData,TargetTrain);

        Results0 = sim(OurNet,TestingDataClass0);
        Results1 = sim(OurNet,TestingDataClass1);

        ErrorsA(Test,TestSerie)  = sum([Results0>0.5]);
        ErrorsD(Test,TestSerie)  = sum([Results1<=0.5]);
        ErrorSum(Test,TestSerie)  = ErrorsD(Test,TestSerie)+ErrorsA(Test,TestSerie)
    end
end
```

Next, results prepared in such a way we can display using the box plot:

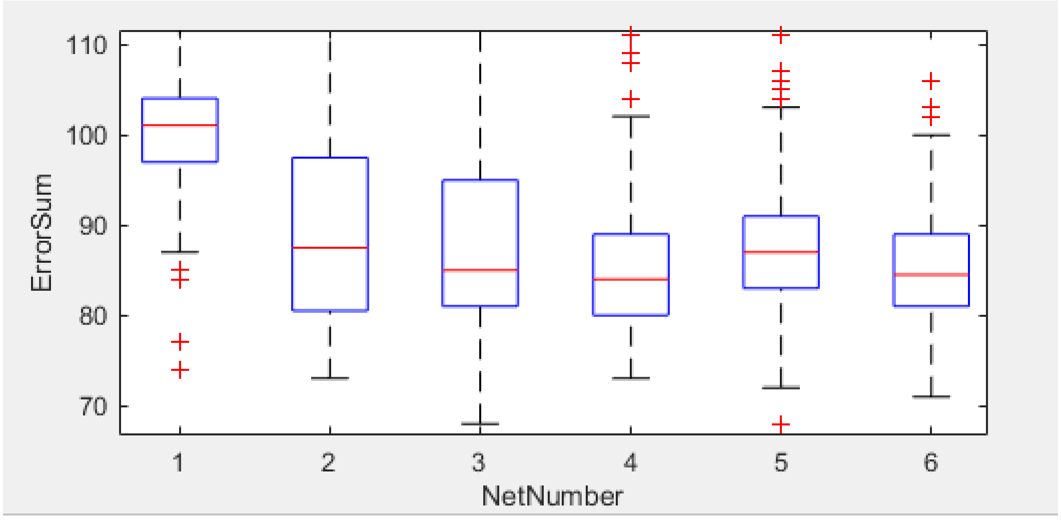
```
figure;   boxplot(ErrorSum);   ylabel('ErrorSum');   xlabel('NetNumber');
```

That may look like this:



In the figure we can see median values (red lines), 1st and 3rd quartile of data and whiskers that spread from min to max of data, with the exception of samples deemed outliers. The outliers are marked separately as red "+" signs.

We can see that in our task the minimum of median value of errors and the highest repeatability of results (the smalles spread) is located roughly around 4th net, i.e net of [8,8] structure. Of course even these results should not be treated as the most reliable, because 10 repetitions of each test does not allow for high confidence of the results. For 100 repetitions a piece (similar range on Y axis) of the results may look as in here:



As we can see, previously we underestimated the spreads, but the 4th net appears to again score the best retuls with high repeatability. Finally, we should check the chosen structure on a testing set and generate a box (just one this time) again. If we obtain a similar box (similar mean etc.) as in optimization that means that the optimization was done properly. Otherwise (especially when net after test was much worse) it might mean that we over-optimized metaparameters (we have a second-order overfit) .

Task 9: For a 2-feature task optimize number of neurons in layers for 1, 2 and 3 layer nets (counting number of hidden layers). For each net under investigation test number of neurons starting from 2 in each layer, up to 20, with 2-neuron step. The number of neurons in each layer should be the same at all times. For each configuration do 10 repetitions of intialization-training-testing procedure, show boxplots of results. Choose the net that you find the best for the task.

Task 10: For a 2-feature task compare results obtained by the net with optimal values of metaparameters, the best kNN classifier, "manual" configuration of a decision tree, and finally, a decision tree from 3rd instruction trained using a genetic algorithm. Remember, that for a non-deterministic methods (MLP and GA-based decision tree) you should provide statistical results .

Note! In order to use the classifier from 3rd instruction in this task you need to map dataset from this instruction to form consistent with data TR_DATA and VA_DATA used in 3rd instruction, i.e. to matrix form in which first column refer to class label while the rest of the columns refer to selected features.

Task 11: Repeat task 9 for 4-dimensional and 6-dimensional feature spaces (that is: optimize net size to these classification problems). Finally, compare the best configuration in each net depth and each dimensionality on one boxplot (that is: First box should provide test results of the best 1-hidden-layer net on 2D dataset, 2nd box: Best 2-hidden-layer net on 2D dataset, …, final box: Best 3-hidden-layer-net on 6D dataset (9 boxes in total, for 3 depths and 3 dimensionalities). Finally, compare the obtained results with kNN classifier.

Additional tasks:

Task 12: Check how efficiently net can transfer knowledge between different datasets: Test the net trained on *Normal* dataset can classify data from *W1* dataset. Check if using more samples from *Normal* dataset for training (all the 600 samples) allows for higher classification efficiency on W1 data? What happens if training dataset would consist of the complete *Normal* and complete *W2* datasets (1200 samples in each class)

Task 13: Similarily to optimization of metaparameters (size) of MLP, perform optimization of features. Prepare a program that will propose different features to test. You may do that using 1+1 approach, genetic algorithm and/or a random algorithm. Was the initially chosen set of features truly an optimal one?

Task 14: Using a neural net classify data in a task defined as your **individual** task plus one additional randomly picked letter. Do this task in two ways: first, by passing a class label equal to -1, 0 or 1 respectively for each class, and for three different outputs of the net. The first one "fires" (i.e. returns 1) for 1st class, 2nd fires for 2nd class etc. Finally, prepare three different nets. One for distinguishing one class against all the others (i.e. answering a question "is this class A or something else?"), second net specialized in recognizing class 2, third specialized in recognizing class 3. In which scenario the efficiency was statistially the highest?

Task 15: How many samples do we need to efficiently train the net? How many do we need to efficiently test the net? Lets test the net on final 300 samples and then lets train it on 300, 200, 100, 50 and 10 points. How will the results differ? Now, let us train the net on 300 samples and test it on 300, 200, 100, 50 and 10. Which value: training error and repeatability or uncertainty of test rises faster with reduction of the respective dataset?