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 10:

Shallow classifiers for object recognition

You will learn how to select a subset of image features for classification, you will build your own decision tree and then compare its results with kNN and ANN models. We will also evaluate which of the method steps is the most influential on the final result

Additional materials:

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

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

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 included 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 data are stored in structures named accordingly. For example, data for the letter A (images and pre-extracted features) are in the structure FeaturesA. The dataset includes letters written in various fonts — regular, bold, italic, bold-italic, and uppercase. Additionally, each letter appears in three subsets: **Normal**: base data, **W1** and **W2**: distorted versions

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

To illustrate how to approach your **individual** classification task, we will use the example of classifying the letters A and D. The first step is to become familiar with the data. Start by loading the relevant 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 in Figure 1:

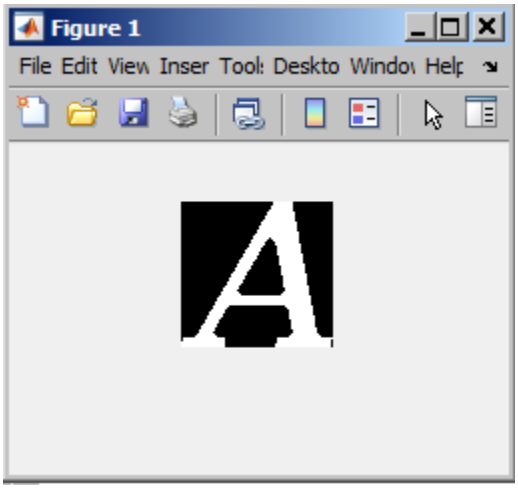


Figure 1 – example of a letter for classification

Task 10.1: 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 that 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 Figure 2. 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%)? Note the green line that shows the concept for such a classification.

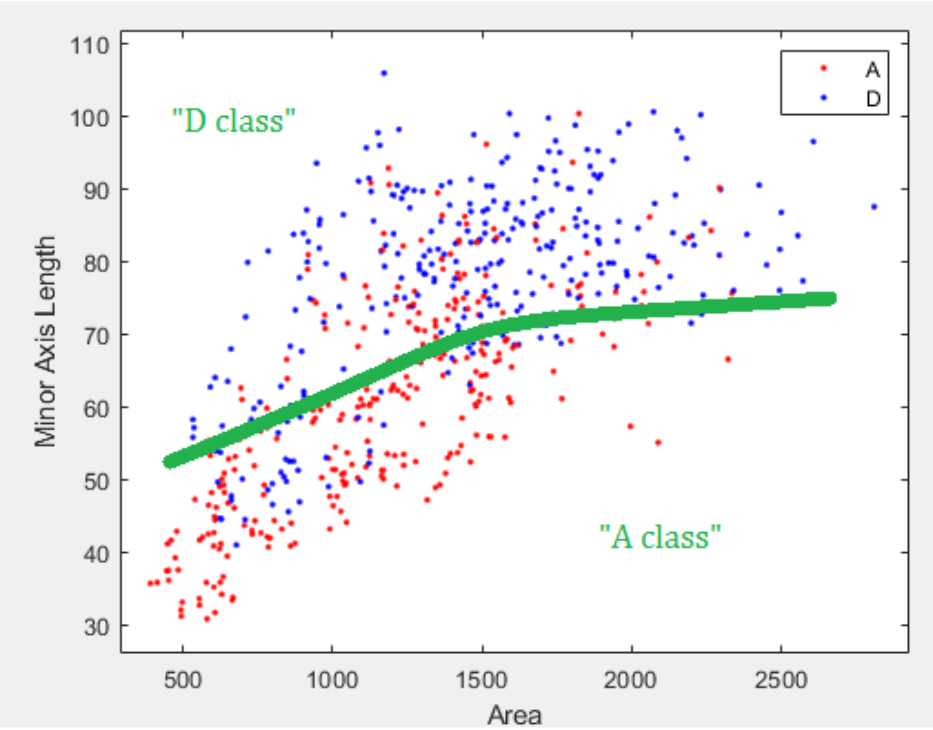


Figure 2 – Feature space consisting of two arbitrary features

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 set of features presented in Figure 3. 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.

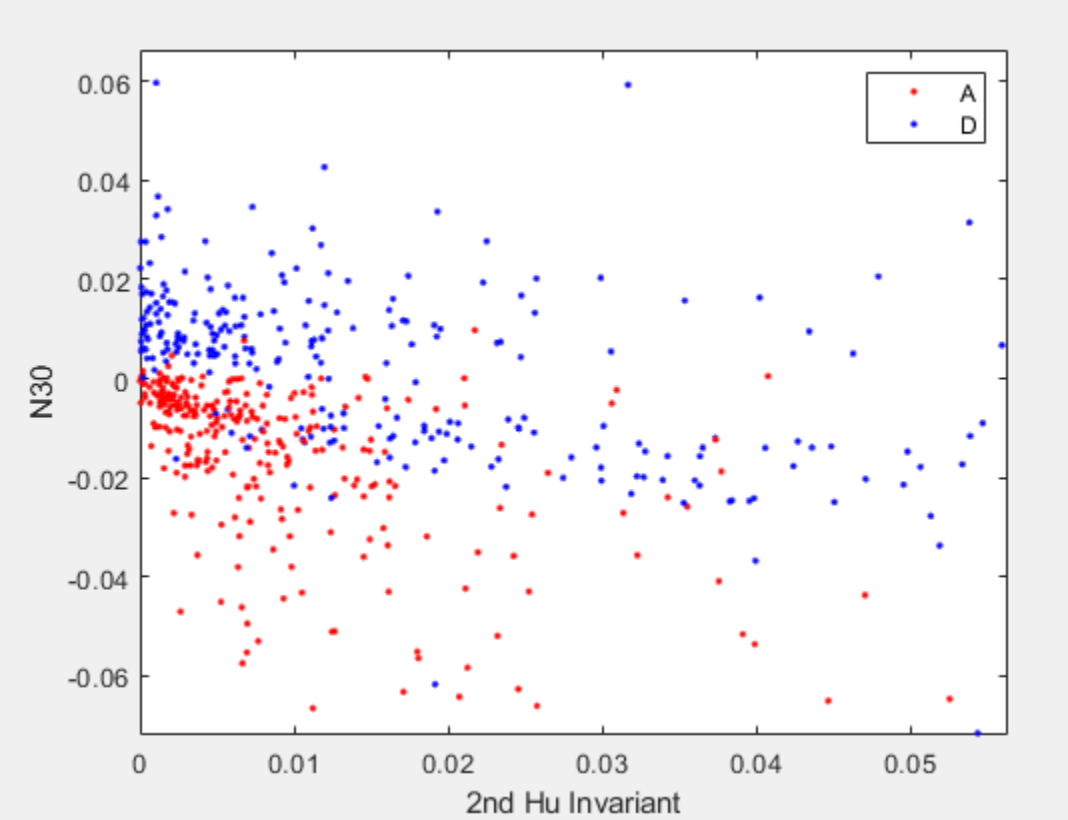


Figure 3 – Feature space allowing for a better class separation than one presented in Figure 2

Task 10.2: 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 obtain 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 Classifier

Based on the selected feature subset, we will now create a set of classification rules to distinguish between the two classes. In our example, we can observe that the normalized central moment N30 tends to be positive for letter "D" and negative for letter "A". Let's build a simple classifier using this observation and save it as a separate function:

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

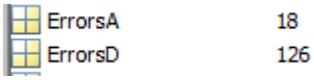
Note that this classifier is structured similarly to the one used before, except that now the classification rule (i.e., the condition on N30) is hard-coded. This means the classifier cannot be trained automatically, but it allows for easy manual adjustment — just by changing the threshold inside the function.

Also, instead of using a general linear separator with three parameters (as before), we're using a simple condition aligned with one feature axis. This limits the classifier's flexibility (fewer degrees of freedom) but makes the classification rule easier to interpret.

Now, let's test how this simple decision tree performs on the training set (first 300 objects in each class):

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

In our example, the classifier correctly recognizes most "A" samples but misclassifies around 42% of "D" samples:



In total, this gives about **76% classification accuracy** (144 errors out of 600 samples). So far, we've only used half of the data — our *training set* — to configure the classifier. Now we'll test it on the unseen portion of the data (samples 301 to 600) to evaluate its generalization:

```
ErrorsA = 0;
ErrorsD = 0;
for k = 301:600
    if(ClassifierDT(A.Data(k).Normal) == 1)
        ErrorsA = ErrorsA + 1;
    end
    if(ClassifierDT(D.Data(k).Normal) == 0)
        ErrorsD = ErrorsD + 1;
    end
end
```

The results show similar performance on the test set, indicating that the classifier generalizes reasonably well. Of course, this is only a simple demonstration. We can expand our classifier to include more conditions or features. For instance:

```
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
```

This classifier uses additional rules and conditions to carve out more nuanced decision boundaries. The graphical interpretation of its decision boundary is displayed in Figure 4.

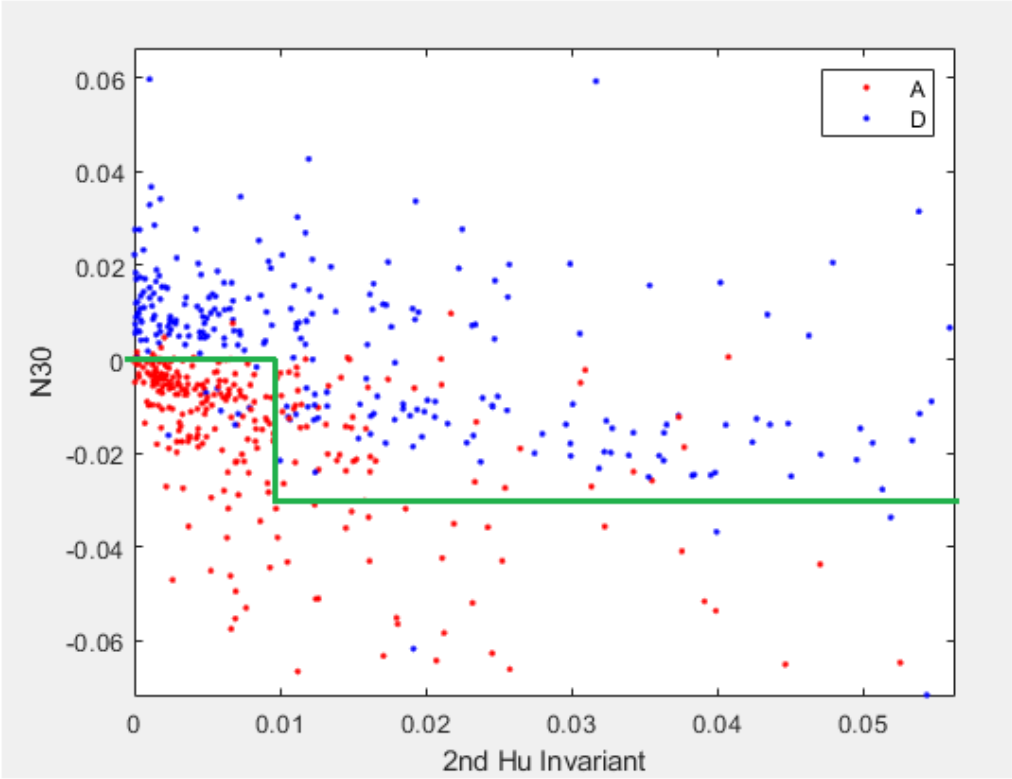


Figure 4 – Multilevel decision tree separation for our data

In this case, the number of errors drops to **93 in the training set** and **115 in the testing set**. We can also visualize where the errors occur the highlighted additions to our code, which should produce result as in Figure 5.

```
ErrorsA = 0;
ErrorsD = 0;
figure;
for k = 1:300
  if(ClassifierDT2(A.Data(k).Normal) == 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

  if(ClassifierDT2(D.Data(k).Normal) == 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');
```

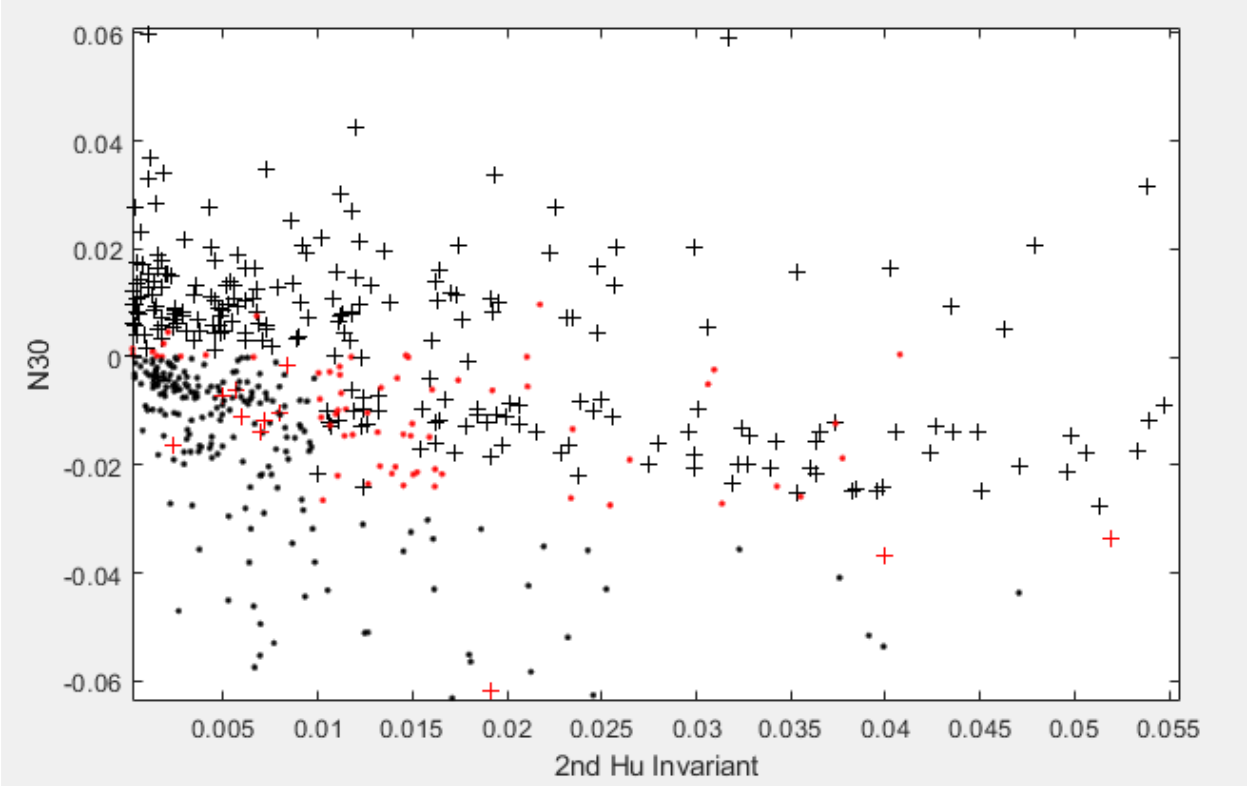


Figure 5 – Error plot

By fine-tuning the rules, we can further reduce errors. For example, the following version of the classifier yields **77 training errors** and **99 testing errors**:

```
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 10.3: 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. Store the results in the table at the end of the instruction.

K-nearest neighbors classifer

To classify objects, we can use the idea of comparing distances in the feature space — the class of an object is determined based on the class of its nearest neighbors. We'll now implement a simple k-nearest neighbor classifier. First, let's construct a training dataset. Note that we again use only half of the data:

```
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
```

Now, we can define the classifier, which for now will use just **one nearest neighbor** (k = 1). Save the following code 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
```

Next, we test the classifier using the second half of the dataset:

```
figure;
ErrorsA = 0;
ErrorsD = 0;

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');
```

This classifier will always achieve 100% accuracy on the training data, because it uses the nearest sample — and in the training set, each sample is closest to itself. However, in our test data, we obtained an accuracy of **82%** (i.e., 104 misclassifications out of 600).

Task 10.4: 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 and the results of both tests in the table at the end of the instruction

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 using distance-based classifiers such as kNN, **normalization** of the feature space is crucial. Without normalization, features with a larger numerical range will disproportionately influence the distance calculation. This can effectively reduce the multi-dimensional space to just one dominant feature.

In our example, the ranges of the two selected features (e.g., N30 and I1) are similar — both span roughly 0.04 units. However, this may not always be the case, especially when adding new features in higher-dimensional spaces.

To avoid this issue, we normalize the data. A simple and effective approach is to divide each feature by a representative value — such as the **maximum** across the training dataset. While this can be sensitive to outliers, alternatives require a bit more complex analysis, relying on determination of data quartiles or its entire statistical distribution. We'll use mean then:

```
% 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 10.5: Normalize your data. Does that affect the classifier efficiency? Store the code so you'll be able to present the results to the LA and store the results in the table at the end of the instruction.

As its name suggests, the "kNN" classifier should be able to use a "k" neighbors 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
```

You can now experiment with different values of k and different feature dimensionalities. Be sure to keep the **feature normalization** consistent across all tests.

Task 10.6: 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 table at the end of the instruction. How well our classifier scales for higher-dimensional spaces? Does the increase in k affects the classifier in any consistent way? Store the results in the table at the end of the instruction.

Multilayer perceptron

We already know how to use MLP networks for classification tasks. Let's return to the code from the previous instructions and adjust the structure of the data matrices to work with MATLAB's implementation of MLPs. We can either classify new points one by one — as in Task 4 — or arrange the testing samples into a matrix and pass them to the network all at once. The latter approach is preferred, as it is much faster and will make subsequent tasks significantly more time-efficient.

Task 10.7: Based on the chosen feature sets used in task 6 test your 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 and save the results in the table at the end of the instruction.

ANN metaparameter optimization

We have now reached the final step of our process. While it would be practical to run one of the selected metaparameter optimization procedures from before to save time and increase the likelihood of improving the result, instead we will take a different approach. We will statistically evaluate which metaparameters actually have an influence on performance. This will require significantly more tests and calculations than strictly necessary to find the optimal network structure, but it will help us understand which aspects are truly important and which can potentially be ignored in the future. To visualize our findings, we will use boxplots. Let's say we want to evaluate the influence of network size — and we want to do it statistically. Below is a code skeleton for this task. Pay attention to the highlighted comments indicating where specific elements should be inserted:

```
for TestSerie = 1:6
   NetworkSize = [2*TestSerie 2*TestSerie]; % Number of neurons in all the hidden layers
   for Test = 1:10

   % Here we should have net initialization with defined network size (to reinitialize
internal weights and start training from scratch)

   % Here we should have the actual training of our net on the features from training
dataset

   % Here we simulate our net on the new (testing) data and calculate sum of errors for
a given net run and configuration...

     ErrorSum(Test,TestSerie) = % ...which we then store here.
   end
end

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

everything went well, after completing the code you should obtain an ErrorSum matrix. Using the boxplot command on this matrix will generate a figure resembling the one shown in Figure 6. In the plot, you can observe the median values (red lines), the first and third quartiles, and the whiskers that extend from the minimum to the maximum values — excluding outliers. The outliers are marked separately as red "+" signs.

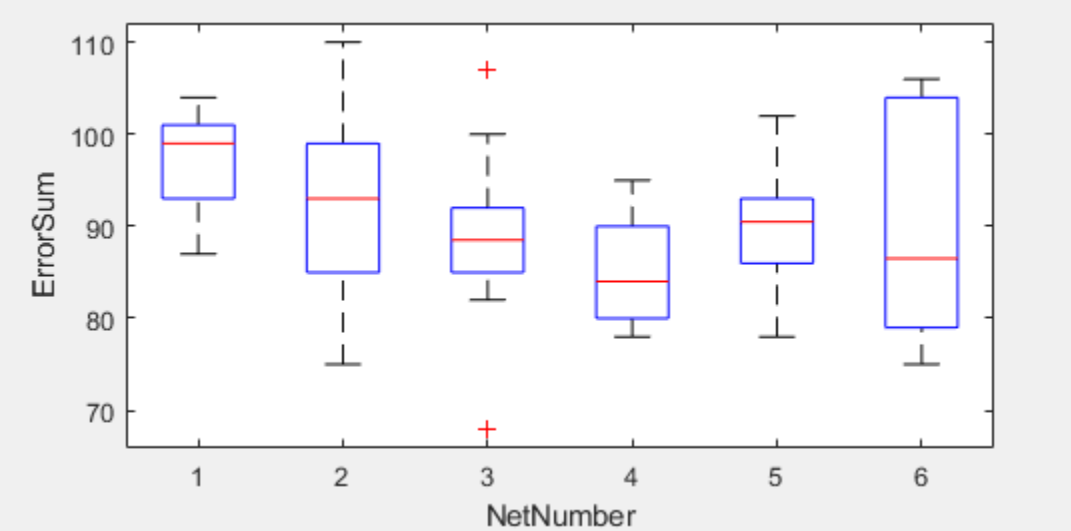


Figure 6 – Boxplots of results obtained for 6 different neural networks. We can compare how different configurations produce different spread of results and have different median values – ranging from 85 to almost 100.

Task 10.8: Lets now use this code structure to evaluate statistically influence of different metaparameters. Check the following metaparameters, evaluating their influence on mean and spread of results:

- Network width (neurons in layers: from 2 to 40)
- Neuron depth (1 – 3 hidden layers for roughly similar number of internal weights)
- Input space shape: different configurations of input features, 2 (multiple), 4 and 6

Display the results in the form of three boxplots, with aligned y axes – to allow for better comparison. Draw conclusions regarding your role as a decision system designer.

Task 10.9: Modify task 8 to include also results of tasks 3 and 6 in a graphical form. Note that these prior assignments should not be evaluated statistically (in this configuration they're deterministic!) – and you already have all the values necessary. Now you need to design visualization that will allow for easy comparison of all the methods.

Additional tasks:

Task 10.10: 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 10.11: 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, and/or a random algorithm. Was the initially chosen set of features truly an optimal one?

Task 10.12: 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 10.13: 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?

Task 10.14: Compare our implementation of the kNN algorithm with matlab function *fitcknn*. Consider both the final accuracy and time-efficiency.

Letters for classification:		
1st feature pair		
2nd feature pair		
3rd feature pair		
	Training:	Validation:
Decision tree, 1st try		
Decision tree, final		
kNN – 2 dimensional	-	
kNN – 4 dimensional	-	
kNN – 2 dimensional, normalized	-	
kNN – 4 dimensional, normalized	-	

kNN: influence of features and neighbors numbers			
	2 features	4 features	6 features
k = 1			
k = 3			
k = 5			
k = 9			
MLP: 3 tries			
Try 1			
Try 2			
Try 3			