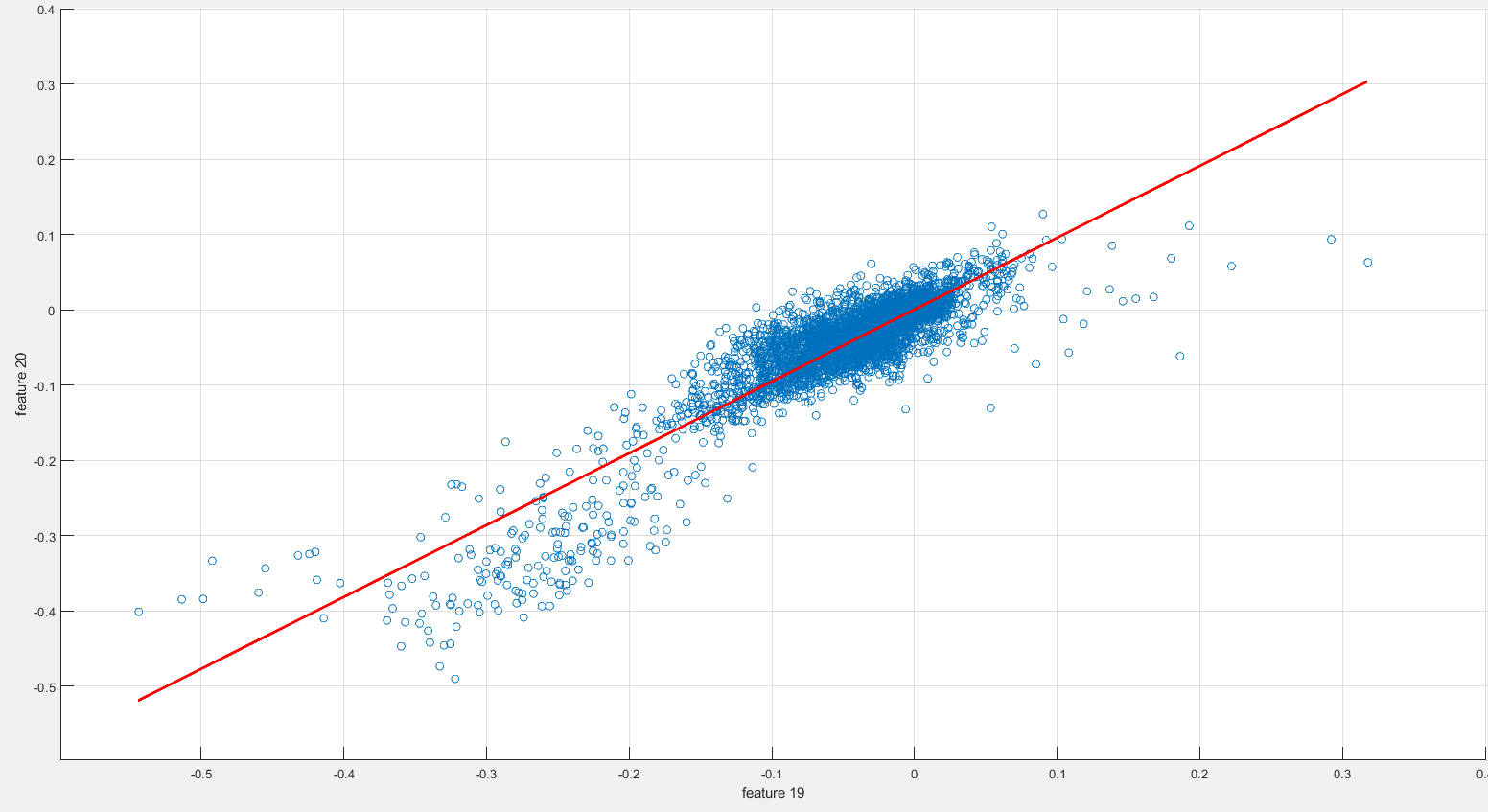
**INF2B Coursework Report Part 1**

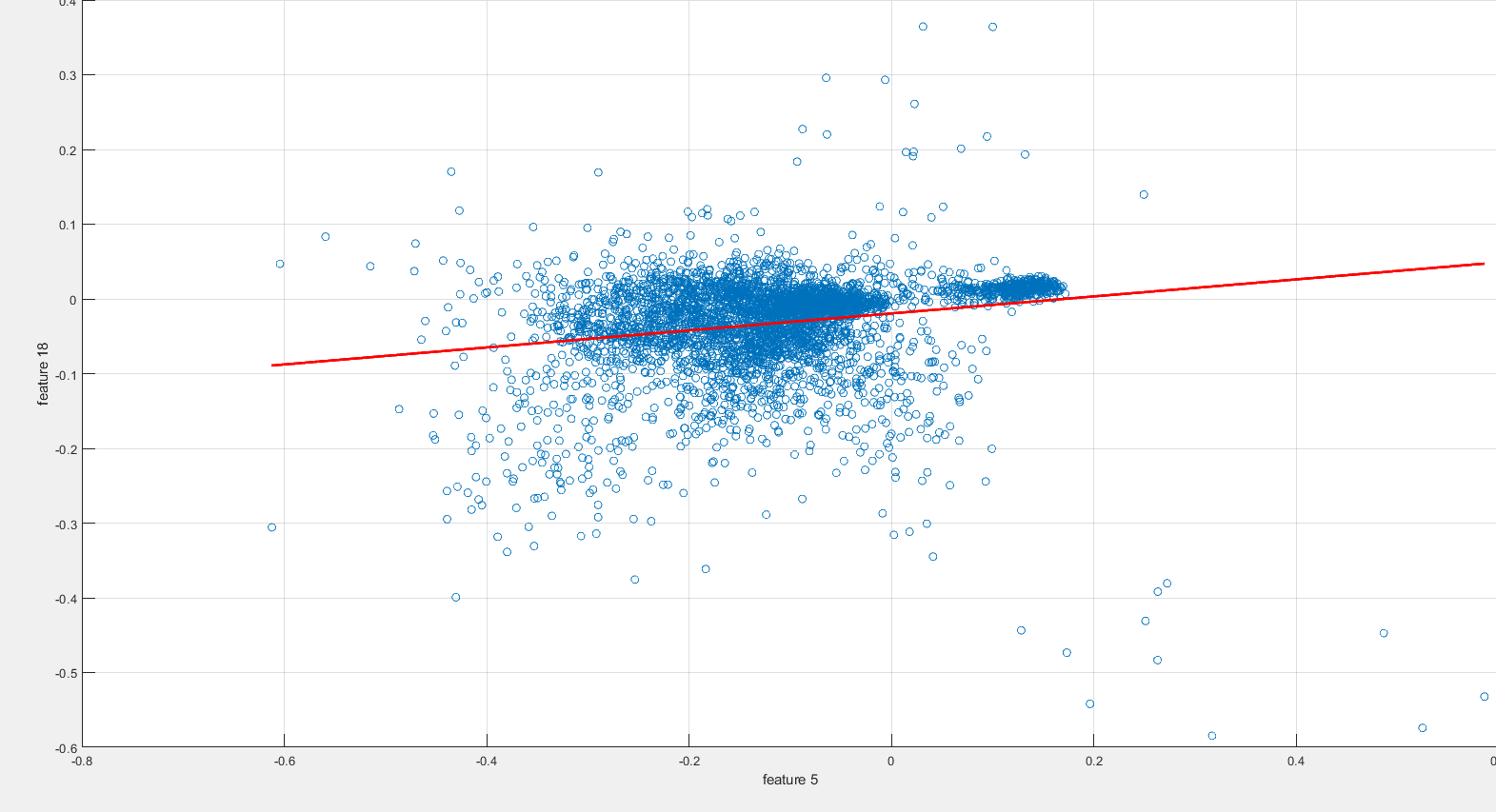
1.2)

(The above image is a covariance plot, which looks very cluttered however and which I will not discuss)

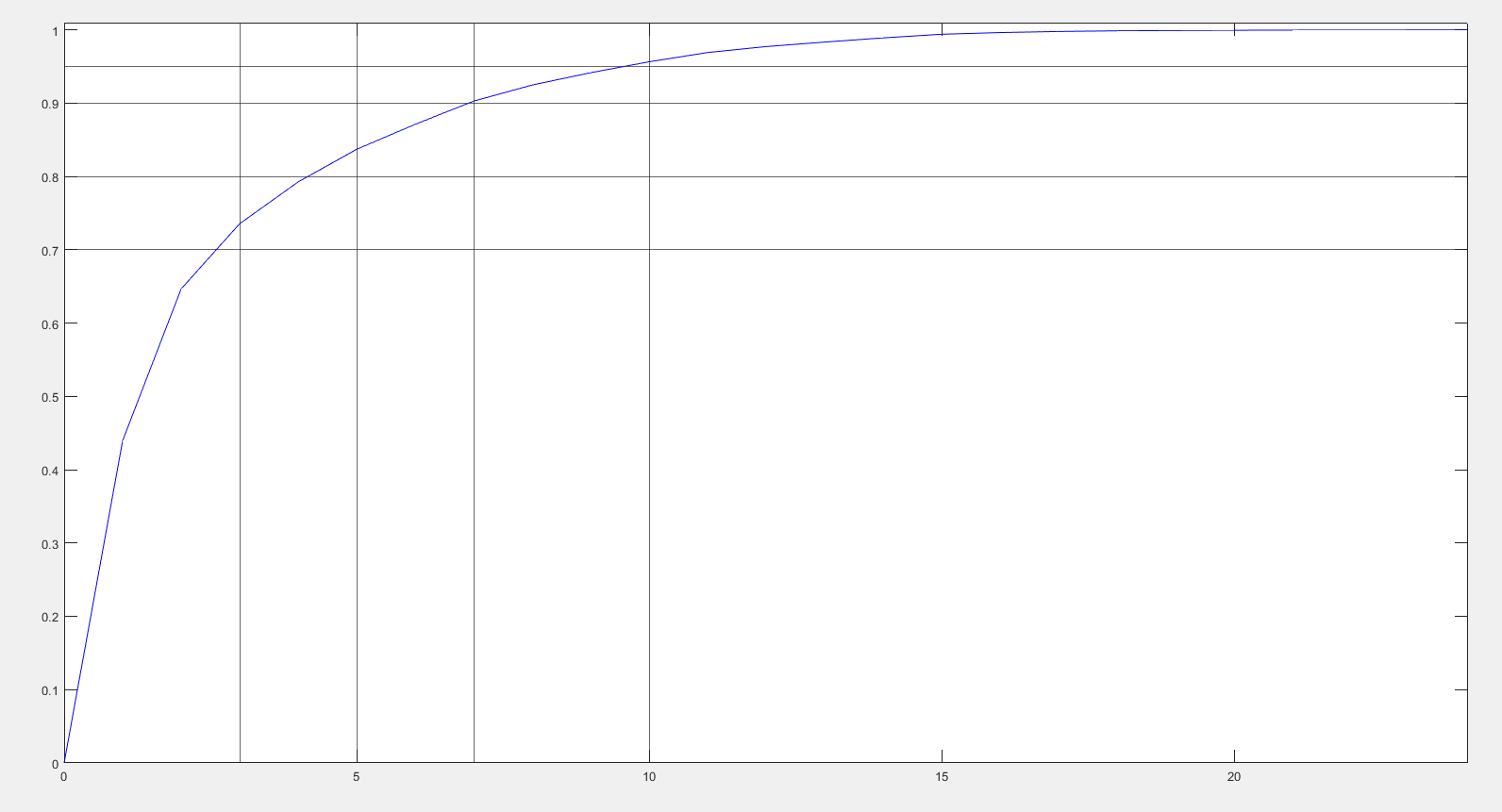
Having computed the correlation matrix for our data set X, I shall now try to analyse particularities of my result.   
The first thing we notice when we look at our correlation matrix is that all entries on the main diagonal are 1. That is entirely expected, since an entry of 1 denotes “perfect” correlation.   
   
All other entries of our matrix are of more interest: A positive correlation matrix entry R\_i,j denotes a positive correlation between the two corresponding features – if a data entry has a value for feature i that is above/below its expected value, feature j tends to exceed its expected value as well. In a way, we can say that the values of the two features tend to “fluctuate together”.   
Good examples for this are e.g. given by the feature pairs 19/20 or 5/6.   
We can analyse their correlation further by drawing a scatter plot depicting all pairs of values the features take, then draw a “line of best fit” that is most suitable of matching the distribution of points we obtain. The positive correlation will manifest in a line of positive slope.   


We see that the values of feature 19 and 20 are both clustered around the value -0.05 and that, when, for a sample, the value of one feature is much lower than that, the value of the other one tends to be low as well – and it goes even further than that. In my data sample, there is not a single example where one of two values is low and the other one large. Furthermore, the “line of best fit” clings to the scattered points extremely closely. Thus, knowing the value of one feature lets us predict the value of the other one rather accurately. This means that the two variables are indeed “strongly” correlated.   
(Note: “Perfect” correlation is the “strongest” possible correlation there can be – knowing the value of the first variable for a sample always lets us predict exactly the value of the second variable for that sample. This also explains why the values on the main diagonal must be 1.)

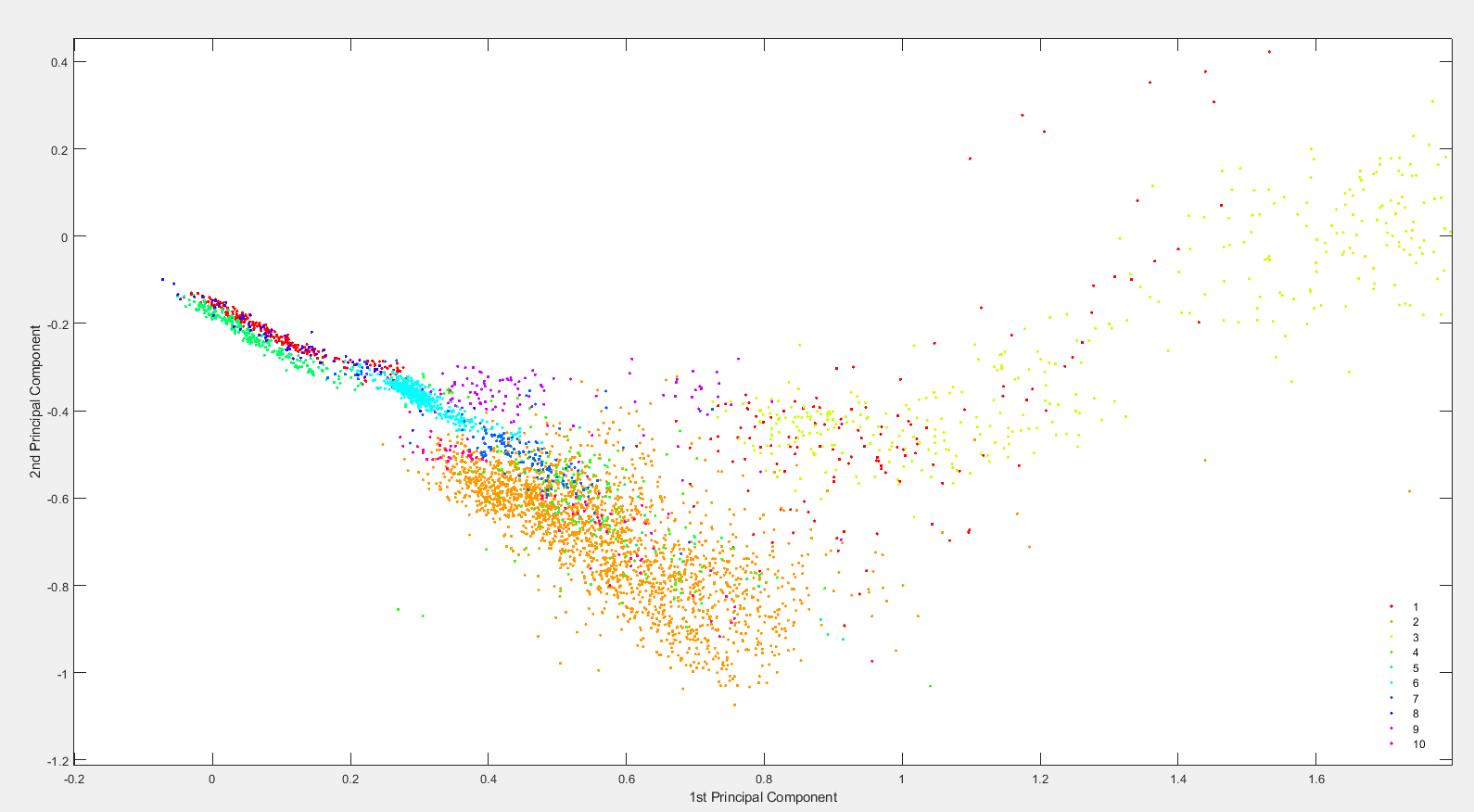
There are other pairs of features for which the “line of best fit” that we get is less instructive, because the individual data points diverge from it heavily, or for which we observe only a weak correlation. Take for example features 5 and 18:



Naturally, I tried to look for pairs of features with a strong negative correlation. That is when I made the surprising discovery that almost all features except feature 1 are positively correlated pairwise, and that feature 1 is negatively correlated to each other feature. That is quite notable and, when properly analysing the database, one would want to understand why this occurs.

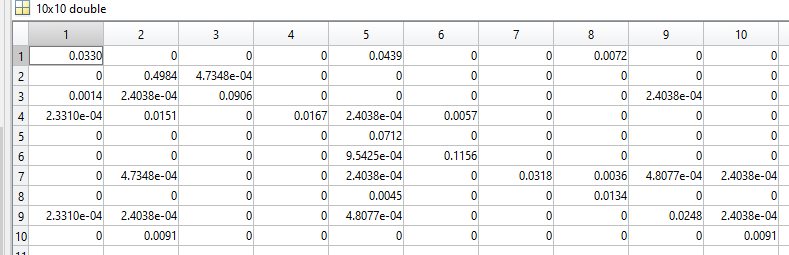
1.3)  
b)

In the above plot, the x values correspond to the number of principal components used, whereas the y values denote the percentage of variance that is accounted for by the according PCA.   
I added horizontal/vertical lines denoting 70/80/85/90% or the corresponding min. number of principal dimensions needed. The graph thus confirms that 3/5/7/10 dimensions suffice to account for the corresponding percentages.

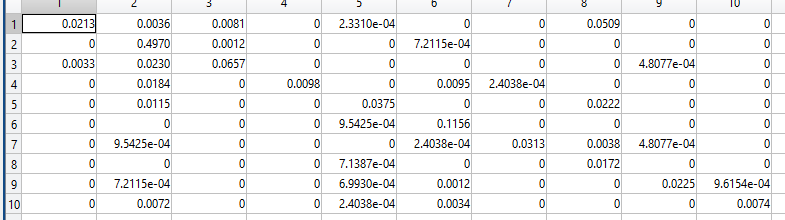
c)  


The above diagram is a scatter plot of all data samples, each reduced to 2 principal components.   
Points of each class/species correspond to one colour, as specified in the bottom right corner of the image.

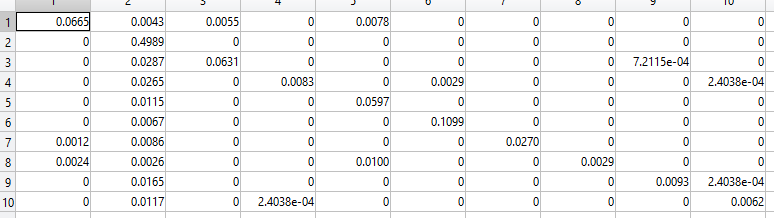
1.4)



By running task1\_mgc\_cv with epsilon=0.01, Kfolds=5 and CovKind = 1, i.e. using a full covariance matrix, we get the above covariance matrix. By adding the relative frequencies on its main diagonal, we find that our classification is correct in 90.45% of all cases. Common error sources include e.g. elements of class 4 or 10 being classified as elements of class 2.

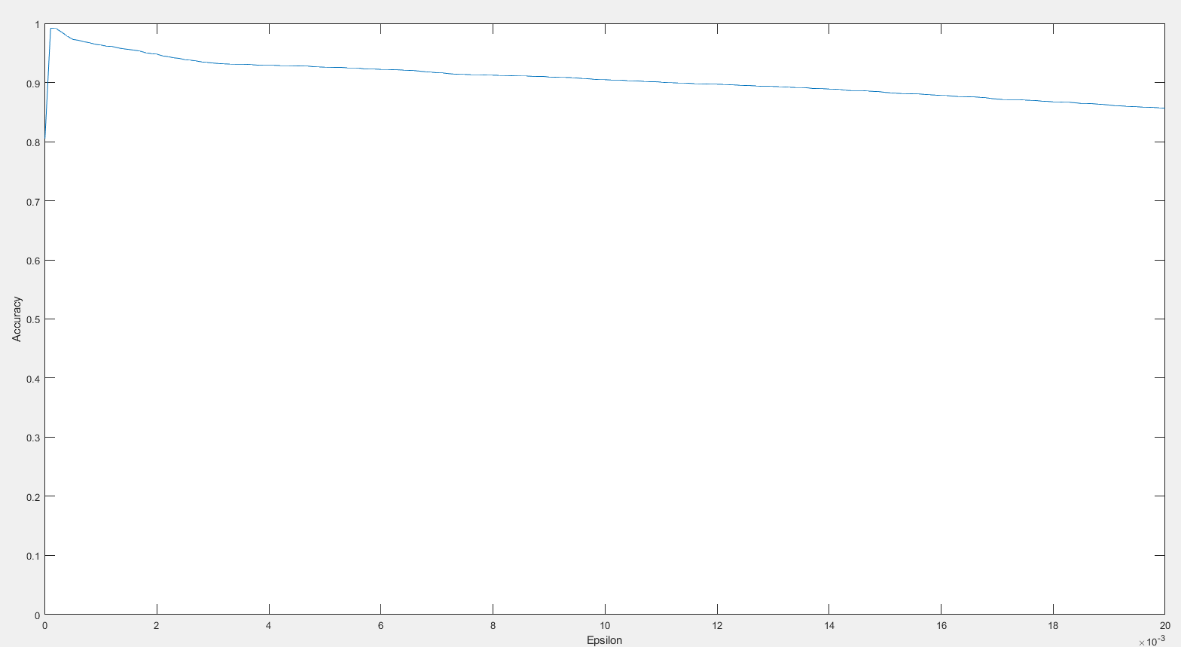


As expected, using a diagonal covariance matrix (i.e. CovKind = 2) makes our classifications less precise. We are correct in 82.52% of all cases.



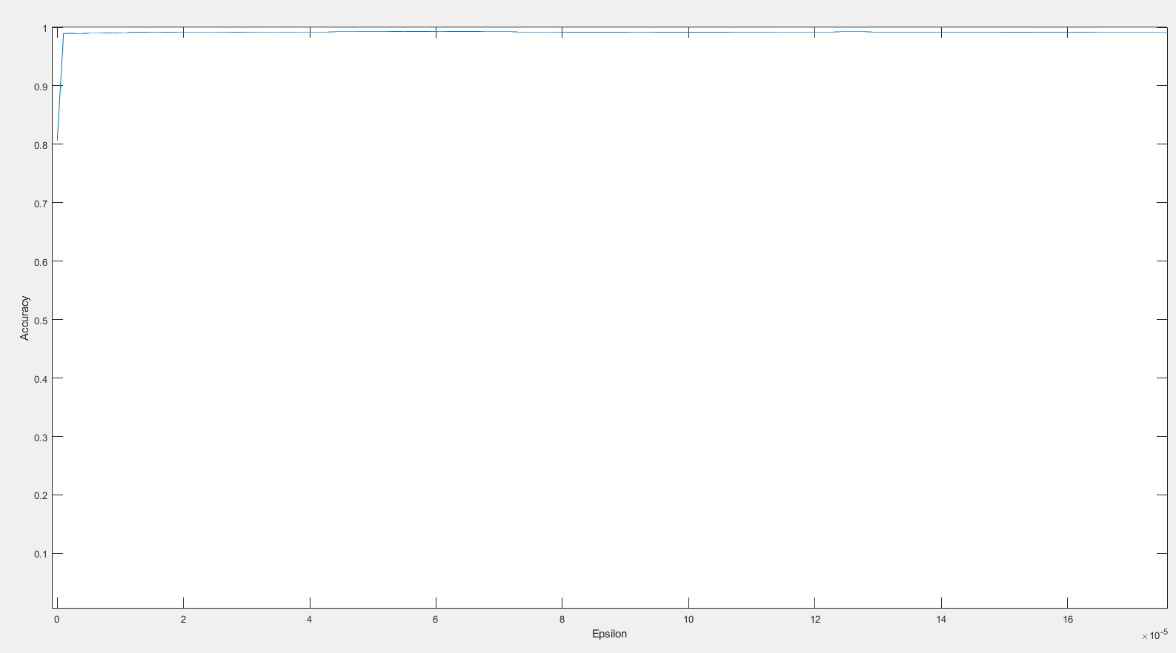
Using a shared (full) covariance matrix yields slightly better results than a diagonal one – the resulting accuracy of 85.18% can however not quite compete with (the computationally most expensive) option 1.

b)  
For the purpose of regularisation, we use a small parameter epsilon, which we multiply the identity matrix by, adding the result to our covariance matrix. I shall now investigate what value for epsilon gives the best result.

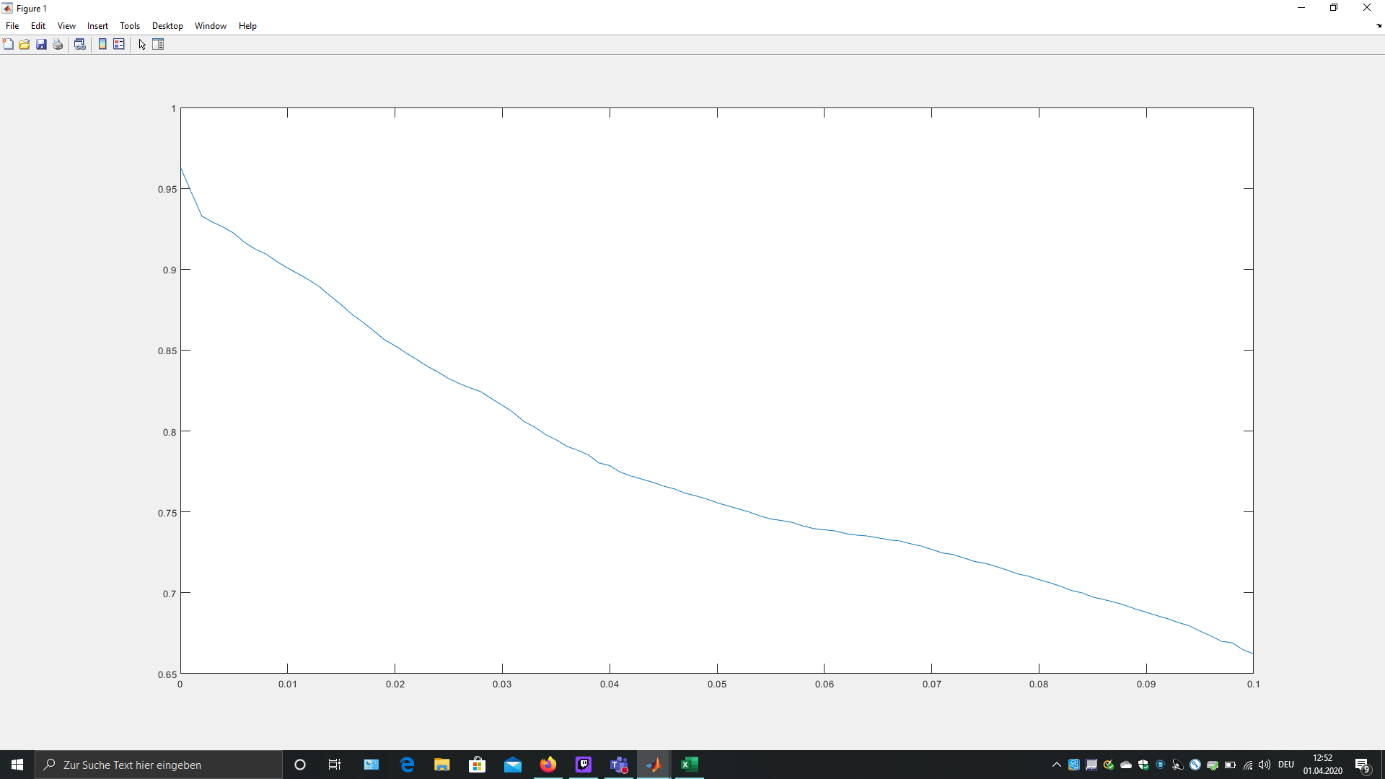
I plotted the accuracy / relative frequency of correct classifications for values of epsilon between 0 and 0.02, using a step size of 0.0001.  
First, I noticed that using an epsilon value of 0 leads to various error messages – this is not unexpected, since we need to apply some regularisation to our covariance matrix to avoid trying to compute the inverse of a singular matrix (a matrix with determinant 0, which has no inverse).   
I then received the following graph:  


From a first glance, we see that there is a small range of epsilon values which work exceptionally well. The accuracy then decreases noticeably for larger epsilon values. In fact, if we ignore the obtained accuracy for epsilon=0, the very first step (epsilon=0.0001) yields the best result, with an accuracy of 99.19%.

By using an even smaller step size of 0.000001, I shall perform an even more fine-grained analysis, to see if we can reach an even better result, using smaller values for epsilon.



It seems that we have found the point where smaller epsilon values do not generally improve the accuracy of our classifications. In this experiment, the best accuracy (99.33%) is reached with an epsilon value of 56/1,000,000. (Notably, the accuracy is better than the ones for the values 1 to 55 / 1,000,000 – implying that there is little point in experimenting with even smaller values)

Larger values clearly give much worse results:   


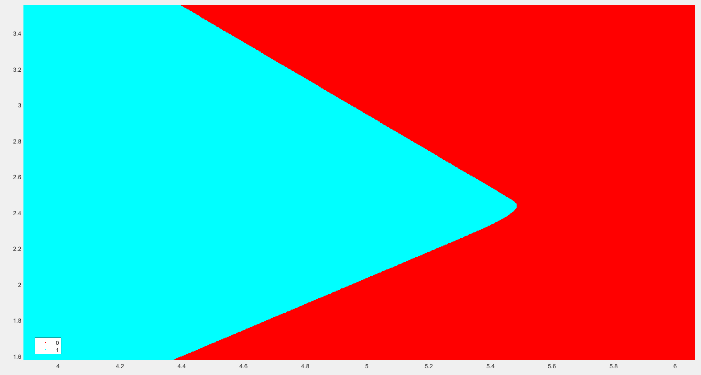
2.3)

First, I will discuss the structure of the required Neural Network.   
We have 2 input values x1 and x2, corresponding to the coordinates of the point we want to classify.   
We will plug them into 4 different neurons with outputs y1 to y4. Each y will be 1 if the point lies “on the inside” of the according weights/line w. respect to the polygon. Thus, we want to classify the point as “1” if and only if y1 to y4 are all 1. (Here, it is important to note that we can only assume this because plotting the polygon shows us that it is convex (i.e. not self-intersecting).)   
We can reach this by adding another hidden layer. We plug y1 and y2 into a neuron with output z1. Z1 equals 1 if y1 and y2 are 1. z1 has the same weights as the neuron with output z2 that takes y3 and y4 as inputs.   
Finally, we can plug z1 and z2 into a neuron with output Y, which has the same functionality as z1 and z2. Then, we have the final output Y.

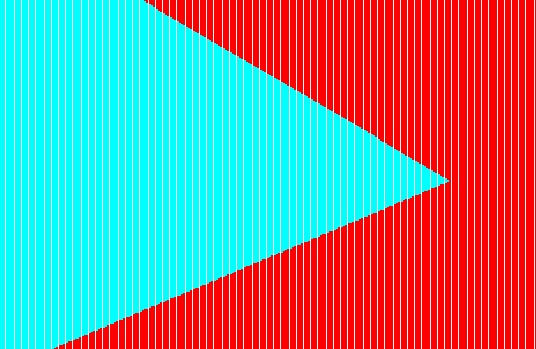
Now, I will explain how I found the weights that realize this.   
For the first 4 neurons, the corresponding weight vectors depend on one of the 4 borders of the polygon, each spanning from a point A to a point B. We compute the border as vector B-A. We then find a vector (w1\*,w2\*) that is perpendicular to the border, compute the corresponding w0\* to get the equation w0\*+t(w1\*.w2\*)=0 and normalize to get the resulting weight vector (w0,w1,w2) as required, making sure that (w1,w2) points to the inside of the polygon in the x1-x2-plane.

The remaining neurons correspond to straight-forward binary AND functions with binary inputs. We have seen in lectures that the line x + y – 1 = 0 will do the job and get corresponding weight vectors (-1,1,1).

2.10)

On first glance, the decision regions look exactly the same. If we look closely however, we observe that the corners of the region corresponding to class 1 look less sharp for the s-Neuron network, whereas the ones for the h-Neurons are perfectly sharp, as expected.  
So the s-Neuron networks can make mistakes for points which are very close to the decision boundaries, especially their corners. And here is why:   
Suppose a point lies just inside of the polygon B, very close to its right-hand corner, i.e. close to 2 boundaries. For those 2 boundaries, the according s-neurons will assign a value just above 0.5. Since we are dealing with a multi-layer network, this uncertainty can accumulate! On the next layer, plugging these two values into an AND neuron may lead to that neuron returning a relatively low value, not much larger than 0.25. This will lead to the relevant point falsely being classified as lying OUTSIDE of the polygon.  


The „round“ right hand corner of the sNN decision region



The perfectly “sharp” corner of the hNN decision region

We counter-act these misclassifications by avoiding such “close calls”, i.e. values around 0.5. This can be achieved by multiplying the weight vectors by a scalar, s.t. the “S” of the sigmoid function graph becomes steeper, thus reducing the proportion of points that are given values which are neither close to 0 nor 1. By making this scalar large, we can make our sigmoid functions approximate the step function with arbitrary precision. For my graphs, I chose a factor of 100. A factor of e.g. 1,000,000 would make the regions completely indistinguishable.