Machine Learning for Robotics I Report

Name: Enrique Rebollo González
ID Number: 5350720

27th December 2021

Lab 1: Naive Bayes Classifier

1 Abstract

This first lab consists in the implementation of a Naive Bayes Classifier that will classify whether the user should or should not play tennis, according to 4 weather criteria: *Outlook*, *Temperature*, *Humidity* and *Wind*.

The database consists of 14 patterns, from which 10 are randomly selected and used to calculate the probabilities of the algorithm (*Training* set). The remaining 4 patterns are later used by the algorithm to check the accuracy of the results obtained (*Test* set).

2 Procedure

For this purpose, two Matlab scripts have been used. A first one containing the function developed *BayesClassifier.m* and a second one used to call that function *BayesClassifierMain.m.* The function developed gets as parameters the *Training* and the *Test* sets and returns the *Error Rate* (only if the *Test* set contains also the targets) and a matrix called *Decision.*

In this *Decision* matrix, each row represents a different instance from the *Test* set, and it is composed of 4 columns as follows:

1. Probability for the instance of becoming a *Yes*, using the formula from Figure 1.

$$\begin{aligned} g_i(\mathbf{x}) &= P(t_i)\,[P(x_1|t_i) \times P(x_2|t_i) \times \ldots \times P(x_d|t_i)] \\ &= P(t_i)\prod_{j=1}^{d} P(x_j|t_i) \end{aligned}$$

Figure 1: Naive Bayes classifier probability formula

2. Probability of becoming a *No*, following the same formula (Figure 1).

3. The decision taken. Which will be the biggest probability from the two previously mentioned (*2* means *Yes* and *1* means *No*).

4. Comparison between the solution obtained for the instance and the correct solution. (*1* means *Correct*, *0* means *Incorrect*). Only computed if the *Test* set comes with solutions.

On the other hand, the *BayesClassifier.m* code is structured in the following steps:

1. The number of columns and rows from each set is obtained at the beginning of the code (*[r1,c1] = size(matrix1); [r2,c2] = size(matrix2);*). Allowing this way the use of sets of different sizes.

   Only the ratio of columns from each set must be consistent: the number of columns from the test set must be equal or one less than the number of columns from the training set. Also, all the numbers present in both matrices must be higher than 1. If any of these conditions are not met an error message is displayed.

2. The probabilities for the computation of the Bayes probability from Figure 1 are obtained:

   - The probability for each value that each variable can take for each class.
   - The general probability for yes and no. Obtained from the *Training* set, computed as "*number of Yes / number of entries*", and the analog for no.

3. The instances from the *Test* set are classified using the probabilities obtained.

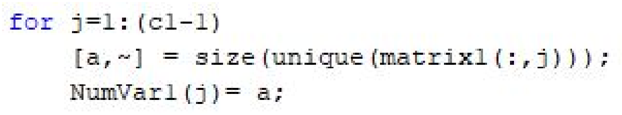
4. If the number of columns from the *Test* set are the same as in the *Training* set (the *Test* also contains the targets), the results obtained in step 3 are checked and the error rate is computed.

3 Challenges

Some problems had to be overcome in some of the steps:

- (Step 2) Get to know how many values each variable can take

  For this, the *unique* Matlab function is used (as shown in Figure 2), storing in the vector *NumVar* the number of different values that each variable can take in the training set. *NumVar* = [3,3,2,2] for our example.



```
for j=1:(c1-1)
    [a,~] = size(unique(matrix1(:,j)));
    NumVar1(j)= a;
```

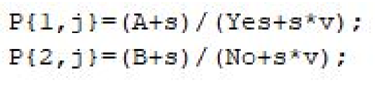
Figure 2: Getting the number of values each variable can take

- (Step 2) How to store the different probabilities

  As the number of values that each variable can take can be different, it cannot be a matrix but a *struct*. It is also needed the use of an auxiliary vector in order to introduce and retrieve the data from each slot of the *struct*.

- (Step 2) Values not present in the training set

  To avoid getting, for values that are not present in the training set, a probability of 0 when computing the formula from Figure 1, a *Laplace Smoothing* is performed.



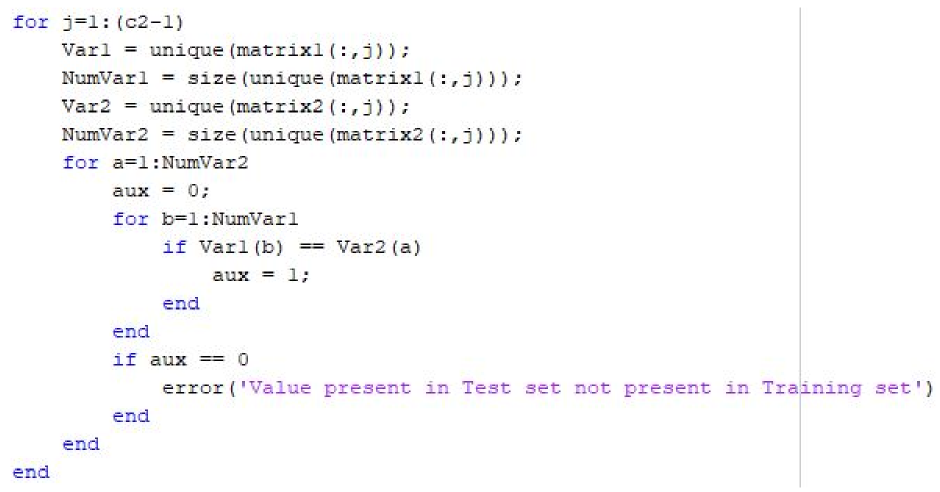
```
P{1,j}=(A+s)/(Yes+s*v);
P{2,j}=(B+s)/(No+s*v);
```

Figure 3: Laplace Smoothing implementation

- (Step 3) Test set contains values that were not present in the training set

  For this, the different values given by the *unique* Matlab function for each column of the *Training* set are compared 1 by 1 with the *unique* values obtained each column of the *Test* set (see Figure 4).

  For each column, the different values and the number of values present in each set are stored in the vectors *Var1*, *Var2* and the variables *NumVar1* *NumVar2* respectively. If a value present in *Var2* is not found in *Var1*, an error message is displayed.



```
for j=1:(c2-1)
    Var1 = unique(matrix1(:,j));
    NumVar1 = size(unique(matrix1(:,j)));
    Var2 = unique(matrix2(:,j));
    NumVar2 = size(unique(matrix2(:,j)));
    for a=1:NumVar2
        aux = 0;
        for b=1:NumVar1
            if Var1(b) == Var2(a)
                aux = 1;
            end
        end
        if aux == 0
            error('Value present in Test set not present in Training set')
        end
    end
end
```

Figure 4: Check if the *Test* set contains values not present in the *Training* set

4 Results

The error rates obtained for four different methods, with and without *Laplace Smoothing* and for different values of the parameter a, are compared:

- Without *Laplace Smoothing*
- With *Laplace Smoothing* and a=0.1
- With *Laplace Smoothing* and a=1
- With *Laplace Smoothing* and a=10

As the test set only has 4 entries, the error rate obtained in each iteration was 0, 0.25, 0.5, 0.75 or 1. This way it was hard to know which was the real error rate of the algorithm, and therefore to compare the results obtained for each model. To obtain more significant results, a loop of 1000 iterations is performed, giving as final result its average.

This way, it was observed that the error rates obtained were similar for the 4 different models, all between 34% and 37%.

5 Conclusion

The algorithm achieves some positive improvements, as without the algorithm the error would statistically be 50%. But the error obtained with all the models is still extremely high, and the adding of the *Laplace Smoothing* does not reduce it. These results must occur because the training and test sets are not big enough to be representative for each class.

Lab 2: Linear Regression

1 Abstract

This second lab consists in the fitting and testing of different linear regression models: both *one* and *multi* dimensional models, and both *with* and *without* offset models.

2 Procedure

1. <u>One-Dimensional without Offset (*Turkish Stock exchange*)</u>

   The slope (w) is estimated from the data available taking the first column as inputs (x) and the second as outputs (t). Following the formula from Figure 5.

$$w = \frac{\sum_{l=1}^{N} x_l t_l}{\sum_{l=1}^{N} x_l^2}$$

Figure 5: Least squares solution without offset

2. <u>One-Dimensional with Offset using *mpg* and *weight* (*Motor Trends*)</u>

   Both the offset w_0 and the slope w_1 are obtained using the *weight* as input (x) and the *fuel efficiency (mpg)* as output (t). Following the formulas from Figure 6.

$$w_1 = \frac{\sum_{l=1}^{N}(x_l - \overline{x})(t_l - \overline{t})}{\sum_{l=1}^{N}(x_l - \overline{x})^2} \qquad w_0 = \overline{t} - w_1\overline{x}$$

Figure 6: Least squares solution with offset

3. <u>Multi-Dimensional with Offset (*Motor Trends*)</u>

   The Moore-Penrose pseudoinverse is used to obtain the *w* vector of coefficients for each dimension (including the offset w_0). Following the formula from Figure 7.

$$\mathbf{w} = (X^T X)^{-1} X^T \mathbf{t}$$

Figure 7: Least squares solution for multi-dimensional regression

4. <u>Task 3: Test Regression model</u>

   In this part the 3 previous models are tested. For this, the data is divided in a training and a test set. The training set contains around 5% of the total entries in the case of the *Turkish Stock* example, and around 20% of the total entries in the case of the *Motor Trends* example.

The average error for each subset is obtained throughout 20 iterations and then compared. This error is computed as the MSE (Mean Squared Error), following the formula from Figure 8.

$$J_{\text{MSE}} = \frac{1}{N}\sum_{l=1}^{N}(y_l - t_l)^2$$

Figure 8: Mean Squared Error

3 Results

3.1 Task 2

In this case, the model explained in point 2.1 was graphically used. First, two subsets of 50 entries (a bit less than the 10% of all the data) were made: one with the first 50 entries of the dataset, and the other one with the 50 last ones. Obtaining a higher value of w for the last period than for the first, as shown on the left image from Figure 9.

Then, to create data subsets from specific periods of time, the data was chronologically divided in 4 subsets. And 4 data sets of 50 entries were randomly made, each one from a different subset. On the right side of Figure 9 the results from 5 iterations are shown. It can be seen that no clear differences can be spotted among the periods other than the first period (*blue*), which tends to have a smaller w than the rest. Also, the second period (*red*) tends to have the highest w.

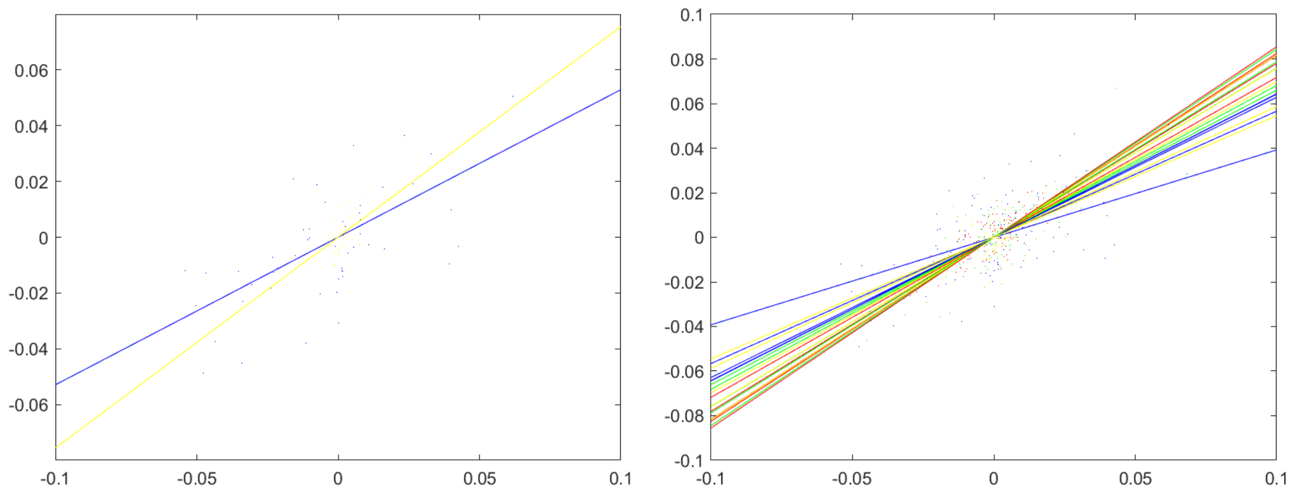


Figure 9: Left: comparison of 50 first (*blue*) and 50 last (*yellow*) entries of data. Right: 5 iteration comparison of data from 4 correlative periods (*blue* - *red* - *green* - *yellow*)

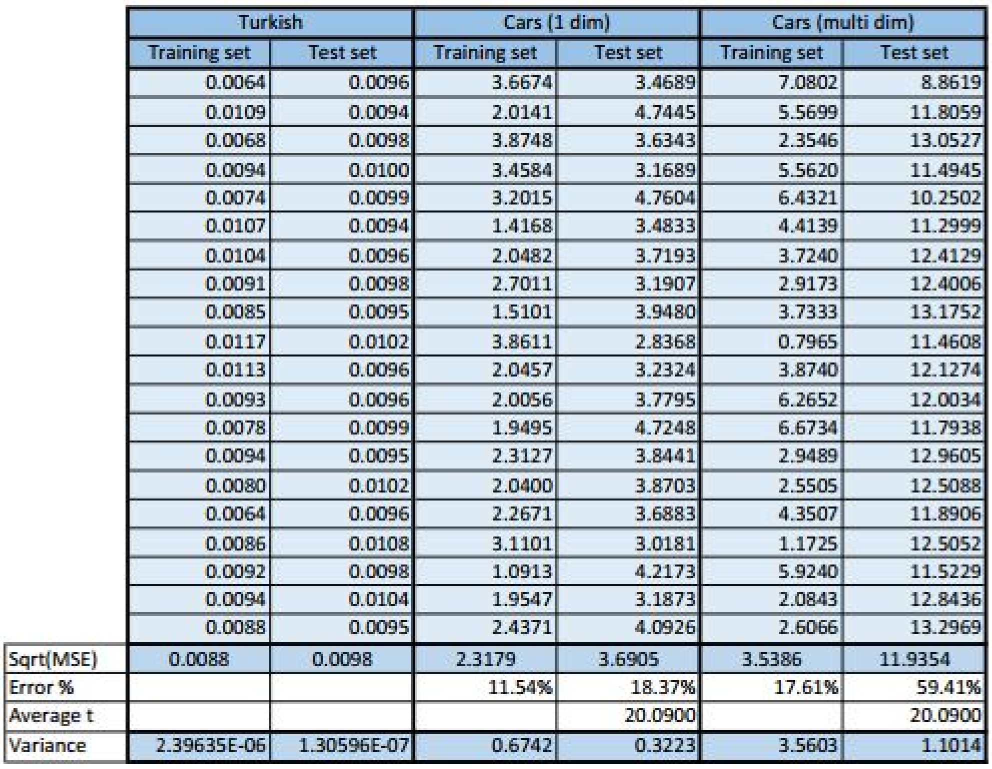
3.2 Task 3

The results obtained from the 20 iterations are represented in Figure 10. It is shown the MSE obtained for the training and test sets for each iteration, along with the average and the variance of these values (*Sqrt(MSE)* and *Variance* in Figure 10). Also the average value of the target (*Average t*) for the complete dataset and the average error obtained (*Error %*) are shown.

Looking at the *Sqrt(MSE)* and *Variance* rows it can be spotted that:

- The average error is higher in the test set than in the training set. This is normal as the parameters that define the model were obtained from the training set. And given that the training set is relatively small, the parameters obtained are rather oriented to the specific instances contained in this subset. If the training set was bigger, even though the parameters would still be taken from the training subset, they would be more representative of the whole dataset. And the errors obtained should be more similar.

- In all the test sets the variance is smaller, even though the average error is bigger. This is due to the bigger size of this subsets, which allows more stable results.



	Turkish		Cars (1 dim)		Cars (multi dim)	
	Training set	Test set	Training set	Test set	Training set	Test set
	0.0064	0.0096	3.6674	3.4689	7.0802	8.8619
	0.0109	0.0094	2.0141	4.7445	5.5699	11.8059
	0.0068	0.0098	3.8748	3.6343	2.3546	13.0527
	0.0094	0.0100	3.4584	3.1689	5.5620	11.4945
	0.0074	0.0099	3.2015	4.7604	6.4321	10.2502
	0.0107	0.0094	1.4168	3.4833	4.4139	11.2999
	0.0104	0.0096	2.0482	3.7193	3.7240	12.4129
	0.0091	0.0098	2.7011	3.1907	2.9173	12.4006
	0.0085	0.0095	1.5101	3.9480	3.7333	13.1752
	0.0117	0.0102	3.8611	2.8368	0.7965	11.4608
	0.0113	0.0096	2.0457	3.2324	3.8740	12.1274
	0.0093	0.0096	2.0056	3.7795	6.2652	12.0034
	0.0078	0.0099	1.9495	4.7248	6.6734	11.7938
	0.0094	0.0095	2.3127	3.8441	2.9489	12.9605
	0.0080	0.0102	2.0400	3.8703	2.5505	12.5088
	0.0064	0.0096	2.2671	3.6883	4.3507	11.8906
	0.0086	0.0108	3.1101	3.0181	1.1725	12.5052
	0.0092	0.0098	1.0913	4.2173	5.9240	11.5229
	0.0094	0.0104	1.9547	3.1873	2.0843	12.8436
	0.0088	0.0095	2.4371	4.0926	2.6066	13.2969
Sqrt(MSE)	0.0088	0.0098	2.3179	3.6905	3.5386	11.9354
Error %			11.54%	18.37%	17.61%	59.41%
Average t				20.0900		20.0900
Variance	2.39635E-06	1.30596E-07	0.6742	0.3223	3.5603	1.1014

Figure 10: Errors computed over 20 iterations

4 Conclusion

It is seen that if a set has more entries it is normally more robust. Therefore, it also tends to have a lower variance. This way, it is important that both the training and test sets have enough entries in order to get representative results.

It has also been noticed that if data comes from different periods of time (as it is the case of the *Turkish Stock exchange*), it may have a slightly different behaviour depending on the period considered.

Lab 3: KNN Classifier

1 Abstract

This third lab consists in the implementation of a *k-Nearest Neighbour* classifier. Focusing on which are the most challenging numbers for the algorithms and how does the size of *k* affect the accuracy.

2 Procedure

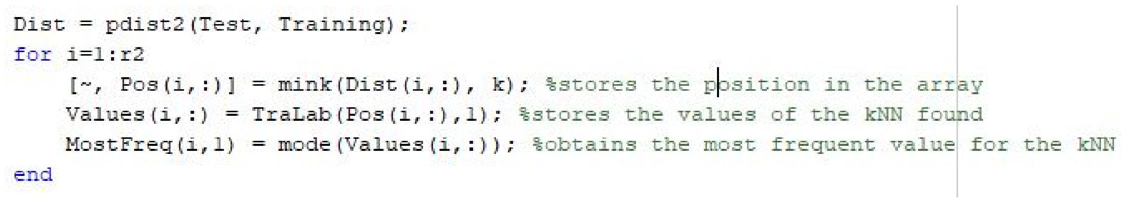
Smaller training and test sets (10% of original size) are created, keeping the same number of instances for each class. For this purpose, the instances for each class are loaded in separated matrices. Equal number of instances are randomly taken from each matrix, added to the reduced training and test sets and then shuffled.

Two scripts are created: one with the algorithm itself *kNN.m* and another one *kNNmain.m* that creates the random smaller test and training sets, as previously described, and calls the algorithm. The algorithm *kNN.m* is structured as follows:

1. Check the number of arguments, number of columns present in the test and training sets, and the value of *k*, as requested in the guidelines.

2. Calculate the kNN neighbour value as shown in Figure 11. First, the *Dist* matrix is obtained with the *pdist2* Matlab function. Each row represents each one of the images from the test set, and each column each image from the training set. Therefore, each row holds the distance between that test image and all the images from the training set.

   The column index from each *k* smaller values for each row is stored in the *Pos* matrix. This column indexes correspond to their position in the training labels vector *TraLab*, from where their values are taken and stored in the *Values* matrix. Then, the most frequent value for each row of the *Values* matrix is stored in the *MostFreq* vector, which represents the final class obtained for that test image.



```
Dist = pdist2(Test, Training);
for i=1:r2
    [~, Pos(i,:)] = mink(Dist(i,:), k); %stores the position in the array
    Values(i,:) = TraLab(Pos(i,:),1); %stores the values of the kNN found
    MostFreq(i,1) = mode(Values(i,:)); %obtains the most frequent value for the kNN
end
```

Figure 11: Algorithm to calculate the chosen kNN value

3. Finally, if the function is called with 5 arguments (also test labels *TestLab*), the error rate is calculated, along with the errors committed for each class. The data is returned as:

   - *Error* matrix containing the class obtained from the algorithm (1st col), the real class (2nd col) and if they both match a 1 and a 0 otherwise (3rd col).
   - *CumulErr* variable holding the Error in the calculations. Computed as *number of errors/total number of instances in the test set.*
   - *E* matrix, containing the percentage of error from each class of the total error.

3 Results

Two main tests are performed:

1. Observe which numbers are more conflictive for the algorithm

   At first, only were computed the global error and the error per real class (e.g. the frequency in which a 1 was detected as something else by the algorithm). See left part of Figure 12.

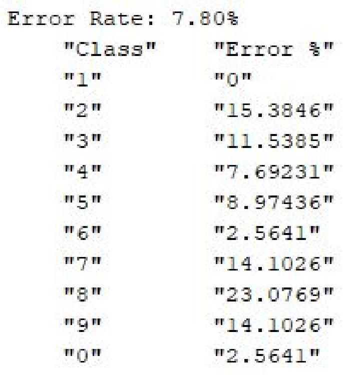
   This method only checked if an instance, being a 1, was actually detected as a 1. But it did not check if instances from other classes were wrongly taken as 1s.

   To tackle this, a more detailed table was developed (see right side of Figure 12).

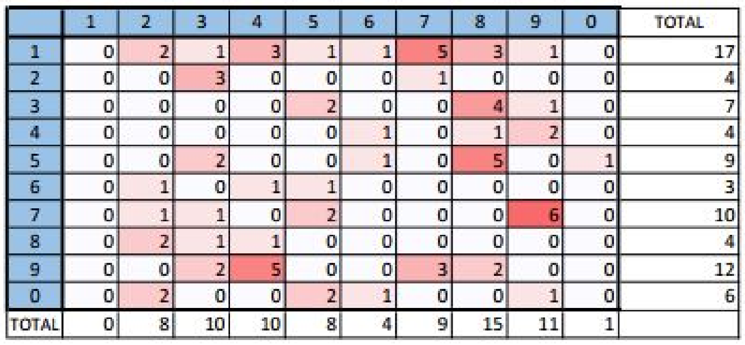
   It is noticed, by checking the last row from the table in Figure 12, that number 1 has, in fact, a very low error detecting 1s as 1s. But, looking at the last column, it is seen that it has the highest error rate of all the classes classifying other numbers as 1s.

   It seems interesting that 7 is confused with 1 but 1 is not with 7. And also that high values are more often confused with low values than vice-versa (9 recognized as 7; 8 recognized as 5; 3 or 7 recognized as 1).

   This could be due to the *mink* function used (third line from Figure 11), that in case of having in a pattern the same number of dominant *Nearest Neighbours* for two different classes, it chooses the lowest number.



```
Error Rate: 7.80%
    "Class"     "Error %"
    "1"         "0"
    "2"         "15.3846"
    "3"         "11.5385"
    "4"         "7.69231"
    "5"         "8.97436"
    "6"         "2.5641"
    "7"         "14.1026"
    "8"         "23.0769"
    "9"         "14.1026"
    "0"         "2.5641"
```



	1	2	3	4	5	6	7	8	9	0	TOTAL
1	0	2	1	3	1	1	5	3	1	0	17
2	0	0	3	0	0	0	1	0	0	0	4
3	0	0	0	0	2	0	0	4	1	0	7
4	0	0	0	0	0	1	0	1	2	0	4
5	0	0	2	0	0	1	0	5	0	1	9
6	0	1	0	1	1	0	0	0	0	0	3
7	0	1	1	0	2	0	0	0	6	0	10
8	0	2	1	1	0	0	0	0	0	0	4
9	0	0	2	5	0	0	3	2	0	0	12
0	0	2	0	0	2	1	0	0	1	0	6
TOTAL	0	8	10	10	8	4	9	15	11	1	

Figure 12: Error rate for k=20: the rows in the table represent the class obtained from the algorithm and the columns represent the real class

2. Observe how does the error rate behave for different values of k

   In Figure 13, it is important to note that only one iteration was done for the calculation of each value of k. So results may not be completely accurate, specially for small values of k.

   Anyway, it can be spotted that the error rate tends to be lower for lower values. But for too low values it may become a bit unstable. Therefore, values between 5 and 10 may be optimal.

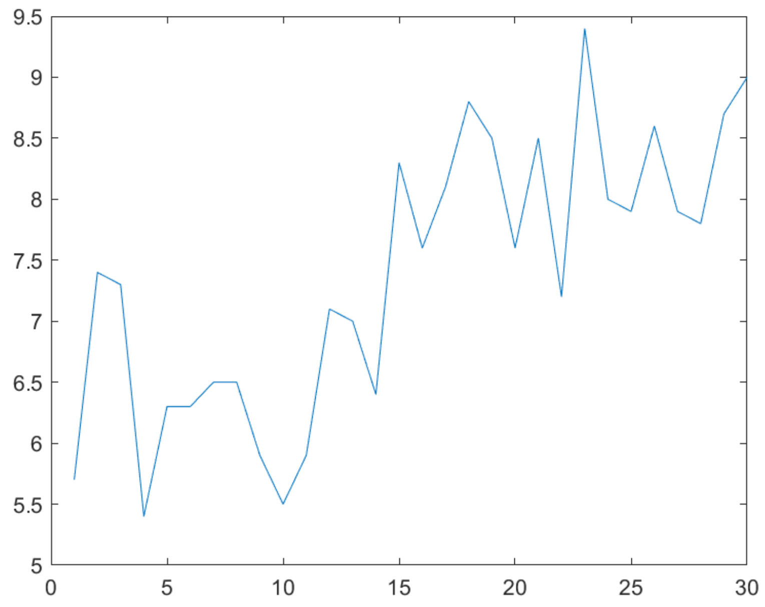


Figure 13: Evolution of the error rate for different values of k (Y axis: Error % / X axis: k)

4 Conclusion

When checking the error rates, it is important to check that a class is correctly being recognized, but also check that this class is not taking also other classes that it should not.

Lab 4: Neural Networks

1 Abstract

The first part of this fourth lab consists in use of the *Neural Network Pattern Recognition* Matlab's app for comparing the results obtained with this method for different datasets and different number of neurons in the hidden layer.

On the second part, an autoencoder network is used for the clustering of the MNIST images of digits already used for Lab 3.

2 Procedure

In the second part only 10% of the data (600 patterns) is randomly taken in each iteration to make the computation faster. Then, two hidden units are used for the *autoencoder*. They represent the x and y coordinates for the representation of each pattern in the final 2D graph.

3 Results

3.1 Part 1

Normally, the larger the number of neurons in the hidden layer, the more precise results tend to be. Anyway, if already good results are obtained for low number of neurons, adding more neurons will not significantly improve the quality of the result (as it happens in the two examples from Figure 14), and computational time may rise consequently.

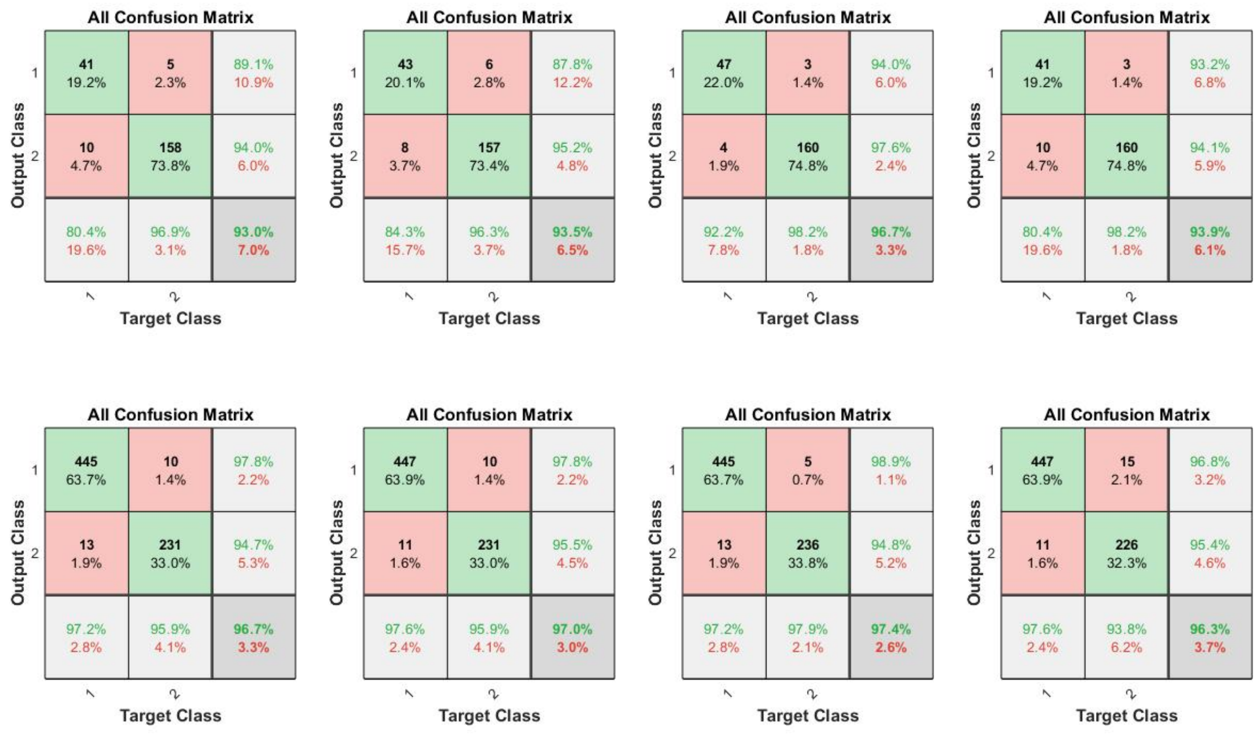


Figure 14: Top row: *Type of Glasses* dataset // Lower row: *Breast Cancer* dataset
From left to right: 2, 5, 10 and 20 neurons in the hidden layer

3.2 Part 2

The pairs tested are the pair of classes that the *k-Nearest Neighbours* from lab 3 confused the most.

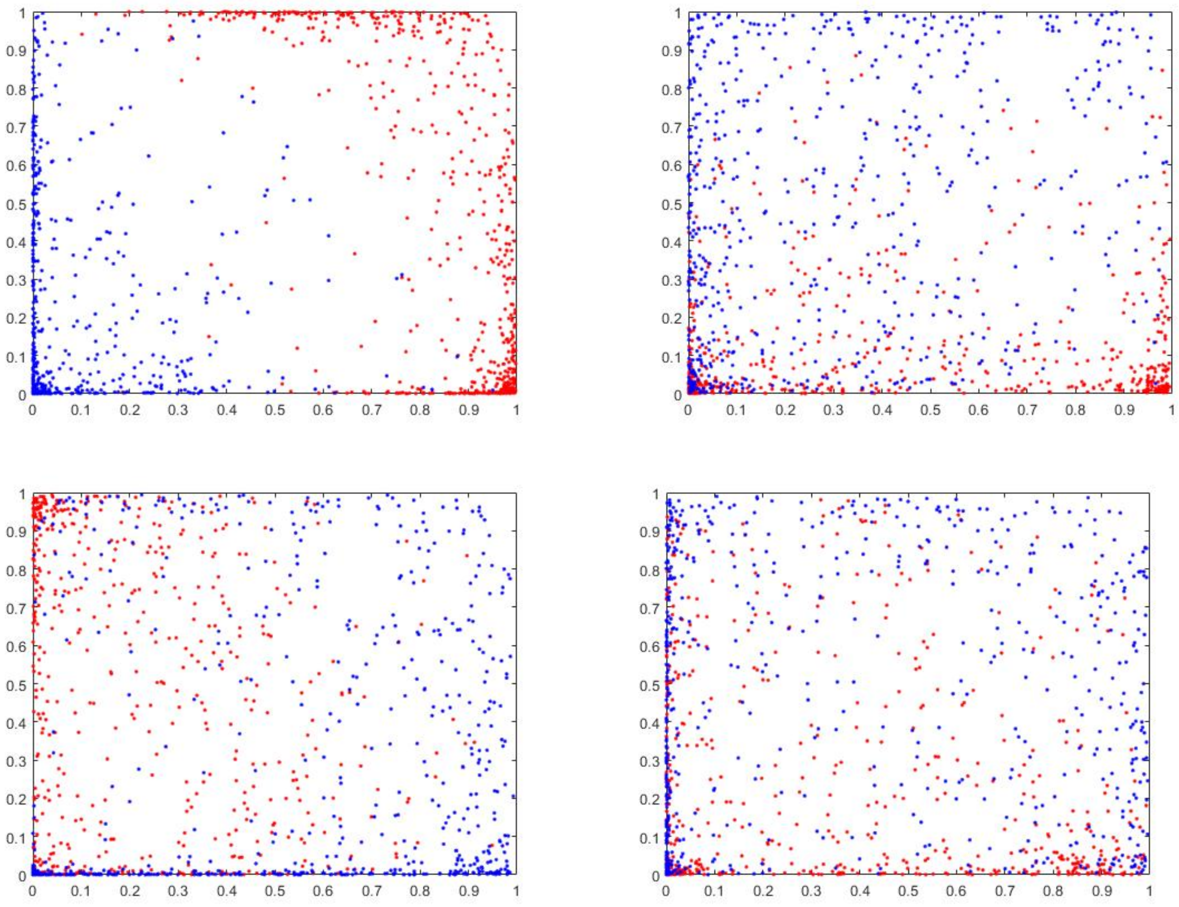


Figure 15: Conflictive pairs of classes: Top row: (1 7) - (5 8) // Lower row: (7 9) - (4 9)

4 Conclusion

The number of neurons and hidden layers present in a Neural Network must be adequate to ensure a balance among the quality of the result and the computational time.

The autoencoder algorithm works very well for non-conflictive pairs of classes like 1-8 or 1-0 (their dispersion graphs were obtained but not included in the report).

Concerning more difficult pairs of classes, like the ones from Figure 15, it works surprisingly well for the pair 1-7. For the other three pairs the results are more unclear. But nevertheless, it is clear that the method works considerably well and can be very useful giving information for telling apart classes.