Klara Schlüter
Machine Learning
NTNU - Department of Computer Science

November 8, 2019

ASSIGNMENT 3 — Programming

2.1 kNN implementation from scratch

The implementation can be found in the Python Notebook `Task1.ipynb`. It can be run cell by cell in order. Relevant explanations are inserted as markup cells. First, the k-NN algorithm using the euclidean metric to measure distances between points is implemented. This implementation is used for regression, where the results are aggregated by a simple mean, and classification, where the results are aggregated based on voting. Then, as described in the assignment sheet, the 124^{th} entry is used as a test sample with $k = 10$. While the regression predicts with an error of 0.2, classification assigns the expected class. The respective 10 nearest neighbours can be seen in figure 1, but are also printed out in the implementation.

2.2 AdaBoost implementation from scratch

The implementation can be found in the Python Notebook `Task2.ipynb`.

Implementing AdaBoost, came at two points where a decision had to be made: First, I noticed that the pseudocode in the assignment sheet and the pseudocode in the slides are different. Comparing them, the normalization of the weights is done in the calculation of the error in the slides, and in the calculation of the new weights in the assignment sheet. Since both do the same in the end, I implemented the version in the assignment sheet. Second, the calculation of a fails due to division by zero if the error of a classifier is zero. I decided to add the smallest positive number to the error to prevent this from happening. This produces infinity for a, and plotting the function that value looked reasonable. The algorithm failed, since the weights couldn't be updated anymore: computation with infinity yielded `nan`. Therefore I decided to use a small, but not the smallest value, which worked. Still, the weights remain equally distributed after that iteration, but that seems kind of what should happen, because an error of zero means the classifier didn't make any mistakes and no samples need to be weighted more then others.

Figure 2 shows the error rate during the training of AdaBoost. Since the first classifier performs perfectly on the training data, the "problem" described in the last paragraph occures: the weights are not altered anymore and the classifier stays the same. The plot shows the error rate for the training data as well as for the test data. For the trainin data, the error is zero, as expected. The final classifier performs less well on test data, but the error also stays constant after the first round, since the classifier does not change

	x1	x2	x3	y
0	6.3	2.7	4.9	1.8
1	6.2	2.8	4.8	1.8
2	6.3	2.5	4.9	1.5
3	6.3	2.8	5.1	1.5
4	6.3	2.5	5.0	1.9
5	6.1	2.8	4.7	1.2
6	6.1	2.9	4.7	1.4
7	6.0	2.7	5.1	1.6
8	6.1	3.0	4.9	1.8
9	6.5	2.8	4.6	1.5

(a) k-NN regression

	x1	x2	x3	x4	y
0	6.3	2.7	4.9	1.8	2.0
1	6.2	2.8	4.8	1.8	2.0
2	6.3	2.5	5.0	1.9	2.0
3	6.1	3.0	4.9	1.8	2.0
4	6.3	2.5	4.9	1.5	1.0
5	6.3	2.8	5.1	1.5	2.0
6	6.0	2.7	5.1	1.6	1.0
7	6.4	2.7	5.3	1.9	2.0
8	6.0	3.0	4.8	1.8	2.0
9	6.5	2.8	4.6	1.5	1.0

(b) k-NNclassification

Figure 1: The neighbourhoods for the k-NN tests.

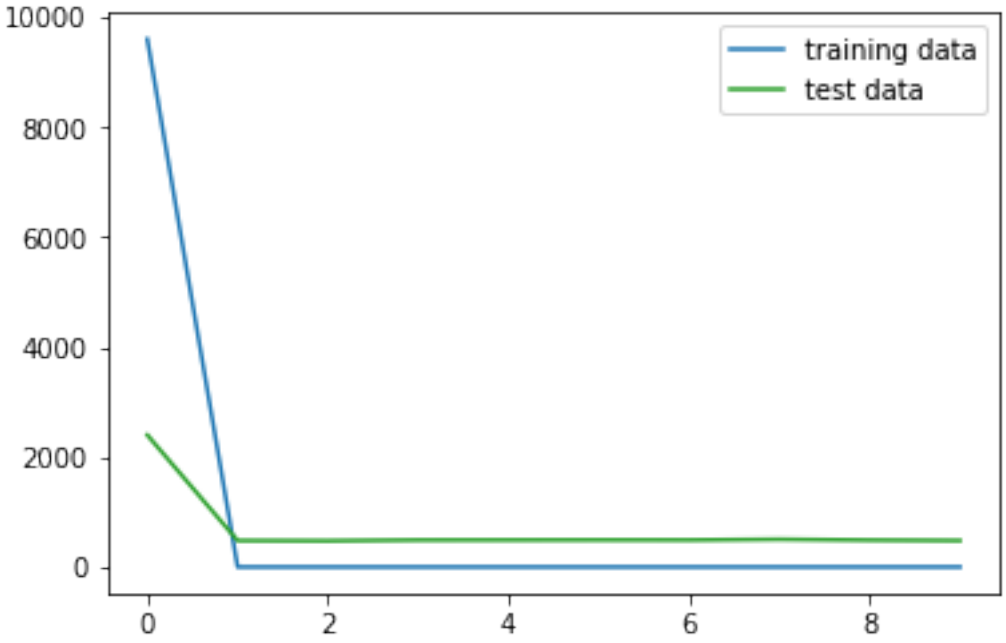

Figure 2: The error rate for iterations of AdaBoost.

anymore. If the first trained classifier performs perfectly on the training data, AdaBoost (and ensemble methods in general) do not make sense.

2.4 Neural Network from scratch

Architecture

The format of input and output for the XOR problem defines the first and the last layer of the network: The input layer has two neurons and the output layer has one neuron. Since the XOR problem isn't linear separable in a cartesian plot, these two layers, describing a linear function, aren't sufficient to solve the problem. Therefore, at least one middle layer has to be introduced. This middle layer needs more than one neuron, since a one-neuron-layer is not able to do anything else than scaling the network output. The minimal architecture according to these restrictions can be seen in figure 3. This small architecture is able to solve the XOR problem, as the implementation (see paragraph **Implementation**) proves.

Hyperparameters

Surprisingly, after some tests, it seemed that the network performed best with a learning rate of 1. Having chosen this, the number of epochs in training had to be determined. A plot of the loss after every epoch, as can be seen in figure 4, helps to find out, when the network achieves the desired loss. The plot shows the loss separately for every instance of the XOR problem. Of course, since the network is initialised randomly, the plot changes a little bit every time it is produced. Even if the networks predictions are already pretty reliable already with smaller values, I chose to train for 2000 epochs. This reduces the probability of the network classifying wrong.

Implementation

I decided to implement the neural network in a purely functional language, namely Haskell. To run it, use the *Glasgow Haskell Compiler Interpreter* (`ghci`). The programm is divided in several modules, please don't change the directory structure. Navigate to the respective directory, load the module XorNN.hs and call `testRun`. This will initialise a small network of the described architecture and train it with a learning rate of 1 and 2000 epochs for the XOR problem. For every instance of the problem, the predictions made by the trained network are printed.

To reproduce the plot shown in figure 4, use the command `trainPlot`.

Submitted by Klara Schlüter on November 8, 2019.

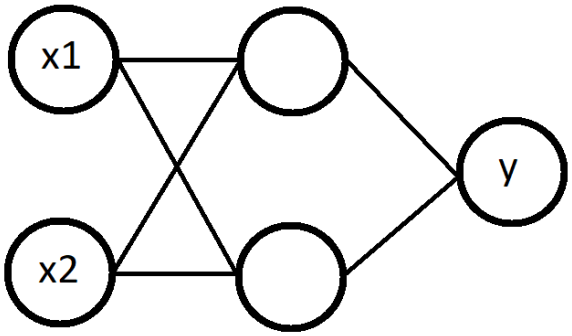

Figure 3: The architecture of the network solving the XOR problem.

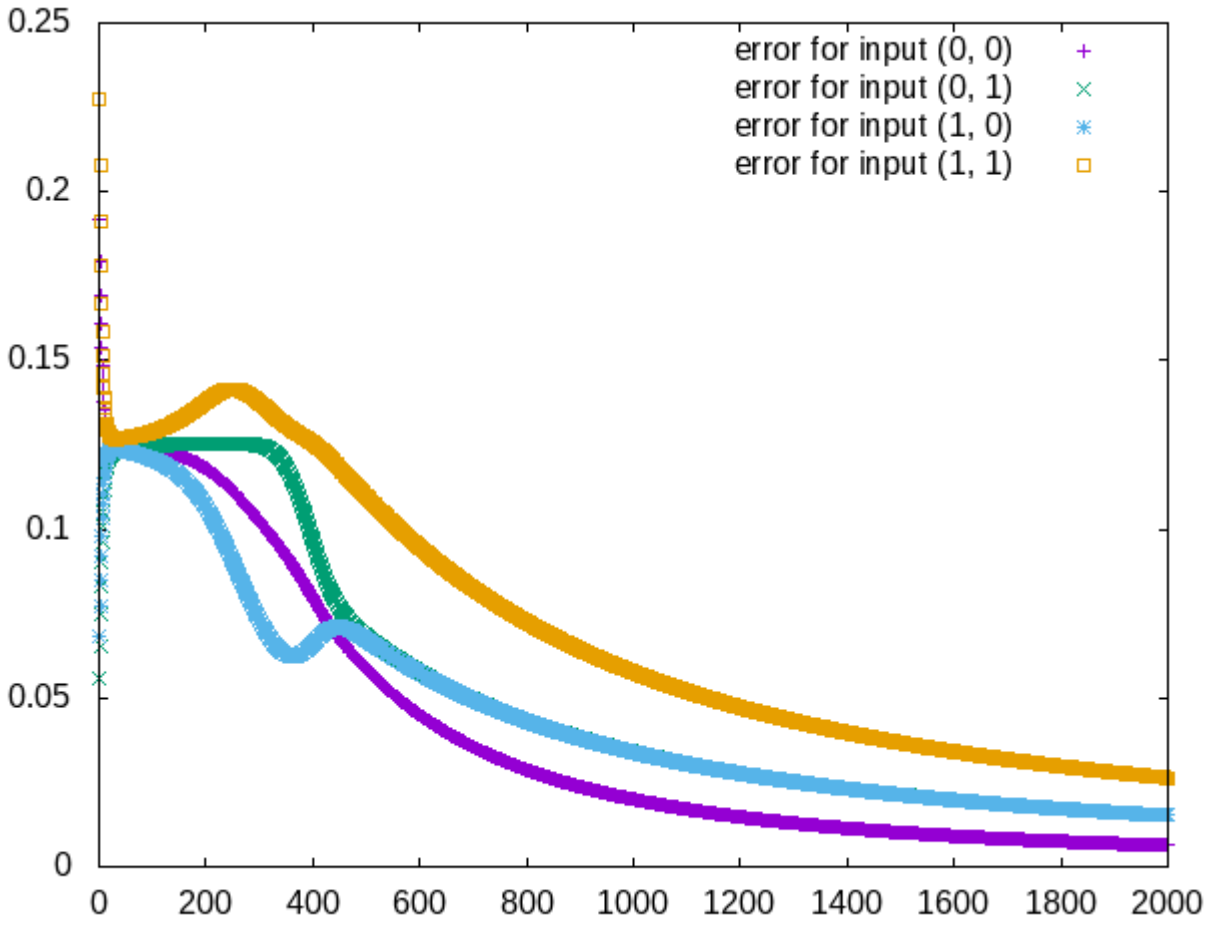

Figure 4: The loss during training the network for the XOR problem.