Klara Schlüter
Machine Learning
NTNU - Department of Computer Science

September 30, 2019

ASSIGNMENT 1 — Programming

Problem 1. Linear Regression

The implementation for this task can be found in the file `Programming-Task1.ipynb`. It is a Jupyter Notebook and can be run by Jupyter with Python 3.

Task 1.2:

We have

- the vector containing the learned bias and weights: $\begin{pmatrix} 0.2408 \\ 0.4816 \\ 0.0586 \end{pmatrix}$
- the mean squared error for the training set: 0.0103
- the mean squared error for the test set: 0.0095

As we can see, the mean squared error for the test set is even smaller than the mean squared error for the training set. Hence, our model isn't overfitting: an overfitting model learns too specific aspects of the training data that can't be applied to other data, so it performs very well on the training data, but not on the test data. Furthermore, it isn't underfitting: if a learned function does not model the data sufficiently, it neither performs well on the training data nor on the test data. The model learned in this task is fitting very good, generalizing beyond the training data very well to predict results for the test data correctly.

Task 1.3:

Figure 1 shows the learned model on training and test data.

The part of the exercise

Does the line you found fit the data well? If not, discuss in broad terms how this can be remedied.

sounds to me as if I should answer this with no and then discuss, but actually, I think it fits as well as a linear function in this dimension could. Maybe not all of the data points are really close to the learned function, but I can't imagine an other linear function for which this would be relevantly better. To have a model fitting the points closer, the model type (hypothesis space) has to be changed (to contain not only linear functions). However, looking for a more complex model might lead to overfitting.

Problem 2. Logistic Regression

The implementation for this task can be found in the file `Programming-Task2.ipynb`. It is a Jupyter Notebook and can be run as such.

Task 2.1.

Linearly separable means (informally), that it is possible to find a hyperplane with all points of one group on one side of the hyperplane and all points of the other group on the other side.

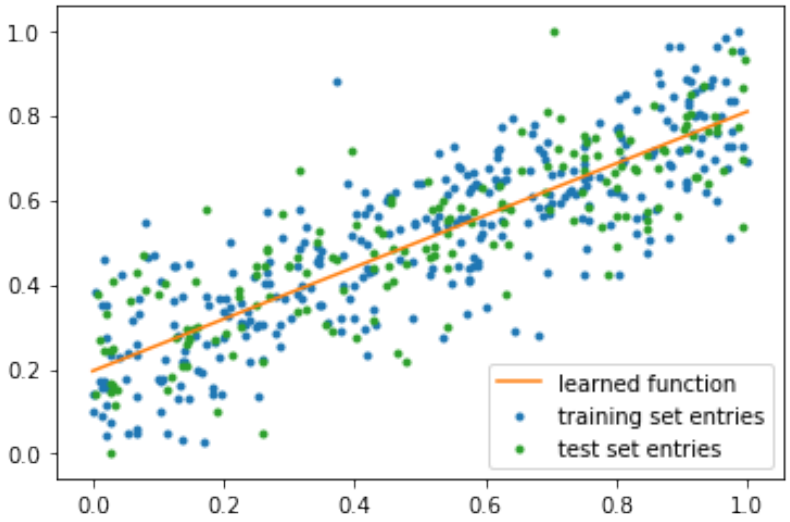


Figure 1: The learned model plotted over training and test data.

In this two-dimensional space, a hyperplane is one-dimensional, a line. In figure 2, we can clearly see, that this is possible, so the data is linearly separable. A logistic regression system is therefore perfect for further predictions.

A training of 1000 iterations validates this thesis: As we can see in in figure 3, The **cross entropy error** is fastly decreasing and converging to zero during the 1000 iterations. As explained in task 1.1., the extremely small difference between the cross entropy error with respect to the training set and the cross entropy error with respect to the test set shows that the obtained model is neither overfitting nor underfitting, but generalizing very well beyound the training data.

After trying different **learning rates** and plotting cross entropy and decision boundary, a learning rate around 0.1 seems to be the best option. For too high rates, the training "jumps over" the optimal point, altering the weights too much. In figure 4a, we can observe this through a cross entropy which is not monotonically decreasing. Too small rates produce a very slow convergence of the model to the optimal function, since the weights are altered in small steps. Figure 4b visualises this by showing again the cross entropy error, also converging very slowly (notice the y axis scale).

Looking for the best values for the **initial** weights, I did some research. All in all, there are two approaches: initialising them randomly or all zero. The easiest way is of course to start with zero. Logistic regression is one of the few models, where this is possible, since the function being optimized is convex. This means that there are no local optima, only one global optimum which we want to find, so by initialising the weights as the origin we can not get stuck in local optima. Furthermore, the weights are updated under slightly different conditions (other than for example in neural networks), so even if we start with all weights having the same value, we can end up with a different value for every weight. Initiating the weights randomly works for every approach, however it is important not to choose too high or to low values, so I chose

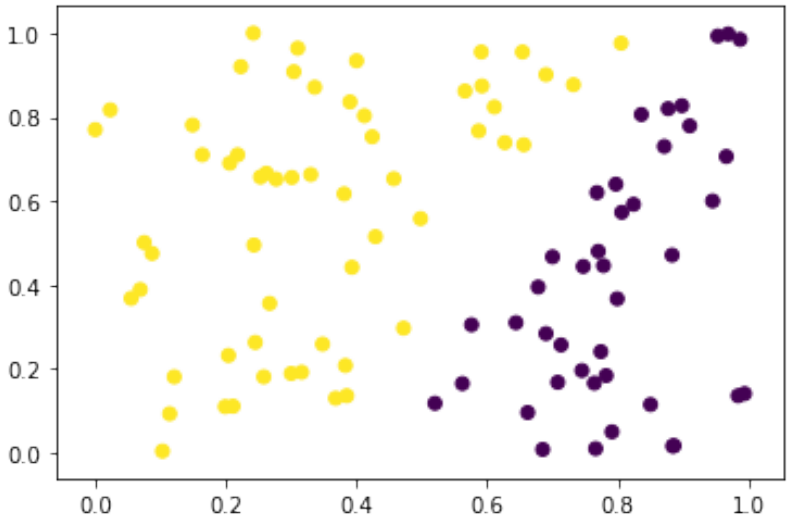


Figure 2: Data points in the first example set, coloured by classification.

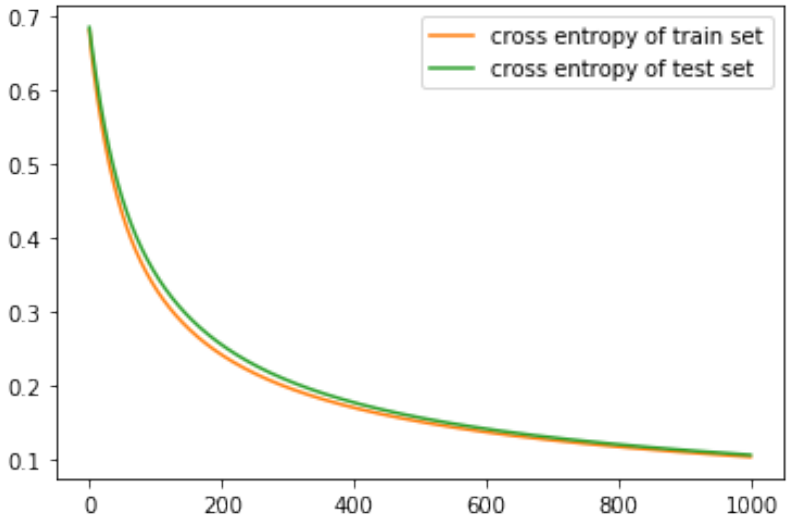
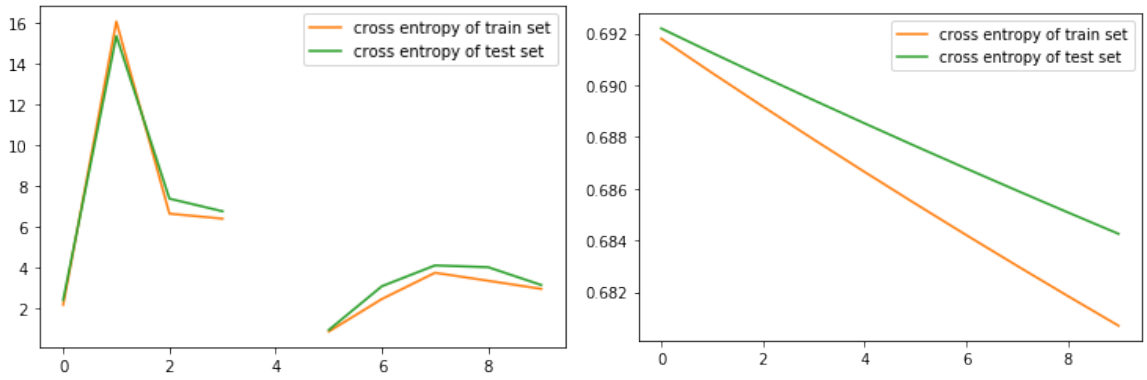


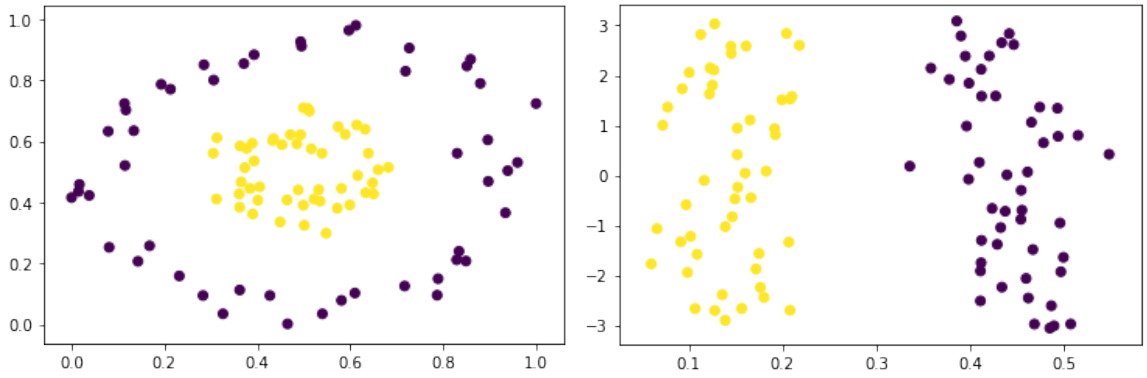
Figure 3: The cross entropy error for both the training and test set over 1000 iterations.



(a) learning rate = 1

(b) learning rate = 0.001

Figure 4: The cross entropy error for different learning rates for 10 iterations of training.



(a) Cartesian plot of data set 2.

(b) Polar plot of data set 2.

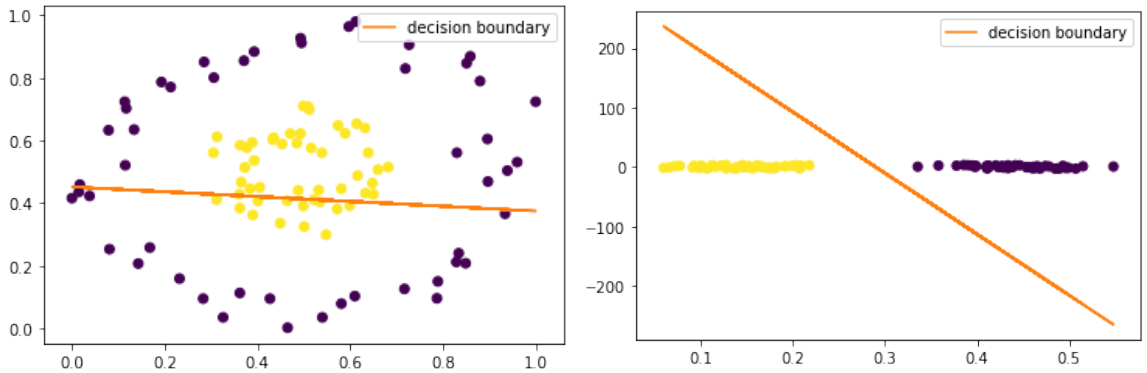
Figure 5: Linear separability of data set 2.

to iniate all weights with zero.

Task 2.2

In the cartesian system, the data set does not seem to be **linearly separable** (see figure 5a). We could draw a line to separate the two categories, but it wouldn't be a hyperplane. The task is to find a system, where the line actually is a hyperplane. Consider the polar system: A specific pole is fixed, preferably in the center of the data. In our case, the pole is the point (0.5, 0.5). In the polar system, the points are not referred to by their absolut x_1 and x_2 value, but by their position in relation to the pole. The horizontal axis in the polar system, the $\rho - axis$, shows the distance to the pole. The vertical axis, the $\theta - axis$, shows the angle to the horizontal axis in the cartesian system. By using the pythagorean theorem and trigonometric functions, we can easily convert the data into its polar representation and then plot again, as it can be seen in figure 5b. In this representation, we can see clearly that the data is linearly separable.

That there is no hyperplane separating the data in the cartesian representation is also a problem



(a) Decision boundary in the cartesian representation. (b) Decision boundary in the polar representation.

Figure 6: Decision boundaries of function learned on the data set.

for logistic regression: since a linear function splitting the data is trying to be learned, it can't succeed on the cartesian data. Figure 6a shows the decision boundary after 1000 iterations of training. For the sake of the argument, I trained on the test data instead of the training data, since the decision boundary visualises the problem better. Since both training and test data are separable by the same line, both fail. In the implementation, you can find the plot for the training data. The decision boundary, a straight line, and therefore the learned function, can not classify the data. There are two ways to alter the logistic regression to correctly classify the points:

- Instead of learning a function representing a straight line, we could change the hypothesis space to the space containing equations representing ellipses (for example the equation $x^2 + 2y^2 + 4$).
- Converting the data to their polar representation and then perform logistic regression in the polar system.

As it can be seen in my implementation, I chose the latter approach, yielding to a perfectly classifying function. The new decision boundary can be seen in figure 6b.

Submitted by Klara Schlüter on September 30, 2019.