Machine Learning for Communications
Neural Networks
Programming Exercises

Marcin Pikus

October 25, 2017

Institute for Communications Engineering
Technical University of Munich
Prof. Dr. sc. techn. Gerhard Kramer

1 Programming Exercises — Set 2

This exercise set regards logistic regression. Note that it is assumed that you have completed the exercise regarding linear regression as we are using the function which were implemented there. Copy the functions which you implemented in the previous exercise in `mlcomm.nn.utils` to the new version `mlcomm.nn.utils` (definitions of the functions are provided for you, you just need to copy your source code inside the function with corresponding name). We are given two two-dimensional data sets `dataset1_logreg`, `dataset2_logreg`. We are going to use the gradient descent (GD) algorithm during all exercises. For each of the exercises you may have to adjust the number of iterations and the step size of the GD to get a good performance. One technique to verify if the GD is behaving properly is to plot the cost versus the number of iterations. During implementation and debugging it is helpful to carefully observe the dimensions of the objects which enter/leave the calculations.

For code brevity, we are going to use slightly different notation than during the lecture. Instead of the separate bias term b in the logistic regression, i.e.,

$$\begin{aligned} z &= \sum_{k=1}^{M} w_k x_k + b \qquad (1) \\ &= \underline{w}^T \underline{x} + b, \qquad (2) \end{aligned}$$

we are going to replace b with w_0 and store it inside the parameters vector $\underline{w}$, just like in the previous exercise. We also need to extend the data vector $\underline{x}$ by additional 1. All the derivations from the lecture should be adapted to this notation (see the third subtask in this section). The extension of the data set with additional 1 will be performed for you in the first exercise by the function `mlcomm.nn.utils.poly_extend_data2D(x)`. The function will take a matrix of training inputs points and extend it with additional vector of ones. That is, if the input is

$$\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(N)} \\ x_2^{(1)} & \dots & x_2^{(N)} \end{bmatrix}, \qquad (3)$$

the function will output

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(N)} \\ x_2^{(1)} & \dots & x_2^{(N)} \end{bmatrix} \qquad (4)$$

- Compute the partial derivatives in the derivation (30) from the lecture notes.
- Derive the formula (32) from the lecture notes.
- After stacking (31) and (32) we get the gradient equation which we will use in this

exercise (the same as for linear regression):

$$\underbrace{\left(\frac{\partial C(\underline{w})}{\partial \underline{w}}\right)^T}_{\text{Notation for programing, } \underline{w} \in \mathcal{R}^{M+1}} = \underbrace{\begin{bmatrix} \frac{\partial C(b)}{\partial b} \\ \left(\frac{\partial C(\underline{w})}{\partial \underline{w}}\right)^T \end{bmatrix}}_{\text{Notation from the lecture, } \underline{w} \in \mathcal{R}^{M}} = \sum_{i=1}^{M} (a^{(i)} - y^{(i)}) \begin{bmatrix} 1 \\ \underline{x}^{(i)} \end{bmatrix} \tag{5}$$

1.1 Logistic Regression

Complete the code inside the file: `log_reg_ex.py`.

- Complete the initialization of w after the line `#random init of w`. Initialize w with samples from the normal distribution.
- Complete the function `mlcomm.nn.utils.lor_cost(w, y, x)`, which computes the cross-entropy cost for the current set of parameters $\underline{w}$ and the training data. The cross-entropy function as in the lecture, i.e.,

$$-y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \text{ for } i = 1, \ldots, N,$$

  is prone to generating not-a-number results. To see this, consider $y^{(i)} = 0$. Then, it is likely that after the training we have $a^{(i)} \approx 0$. The term $y^{(i)} \log(a^{(i)})$ should be 0, but if the value $a^{(i)}$ is small enough to make the logarithm go to `-inf` (because of the exponent in the sigmoid function overflowing) we will get not-a-number result for $y^{(i)} \log(a^{(i)})$. Therefore your implementation should avoid evaluating the terms which we know are 0 but may cause not-a-number result. E.g., if $y^{(i)} = 0$ compute only the term $(1 - y^{(i)}) \log(1 - a^{(i)})$ (which shouldn't cause numerical problems, why?), and if $y^{(i)} = 1$ compute only the term $(-y^{(i)}) \log(a^{(i)})$.
- Complete the function `mlcomm.nn.utils.lor_grad(w, y, x)`, which computes the gradient with respect to w for logistic regression.
- Run the GD algorithm and find the correct values for the number of iterations and learning rate. You should obtain similar plot as below and the cost about 0.1.

1.2 Logistic Regression - non-linear model

In this exercise we work with the file: `log_reg_ex.py`. In this section we will extend our model to include non-linear functions of the input data. This modification greatly improves the variety of the decision boundaries which can be generated.

- Modify the function `mlcomm.nn.utils.poly_extend_data2D(x,P)`.
  Recall that the function used to output

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \ldots & 1 \\ x_1^{(1)} & \ldots & x_1^{(N)} \\ x_2^{(1)} & \ldots & x_2^{(N)} \end{bmatrix}. \tag{6}$$

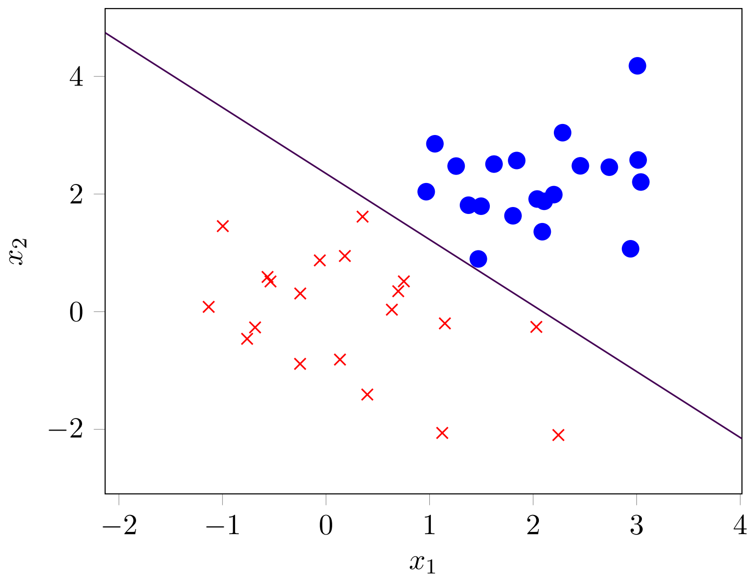

Figure 1: Decision boundary for linear logistic regression.

Now, we would like to perform regression also with respect to non-linear functions of. We will introduce polynomial terms up to degree P. In the two-dimensional case we get more terms because of the cross terms, e.g., for $P = 3$ we have 10 terms: $1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, x_1^2x_2, x_1x_2^2, x_2^3$ and the output matrix will be of size $10 \times M$ with the first row being a row of ones, second row of the samples from X_1, and so on.

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(N)} \\ x_2^{(1)} & \dots & x_2^{(N)} \\ \vdots & & \vdots \\ x_1^{(1)}(x_2^{(1)})^{P-1} & \dots & x_1^{(N)}(x_2^{(N)})^{P-1} \\ (x_2^{(1)})^P & \dots & (x_2^{(N)})^P \end{bmatrix}. \tag{7}$$

Of course, $\underline{w}$ should have now the corresponding number entries, and the data normalization should be performed on the newly generated matrix $\mathbf{X}_{\text{data}}$. P should be a parameter which you can change. Note that for $P = 1$ we have the same model as before, i.e., linear.

- Modify also the wrapper `extension_wrapper(x)` so it uses the newly added polynomial extension with degree P.
- Find the parameter P, the learning rate, and the step size to get the cost function below 0.01. The result may look like in Figure 2.

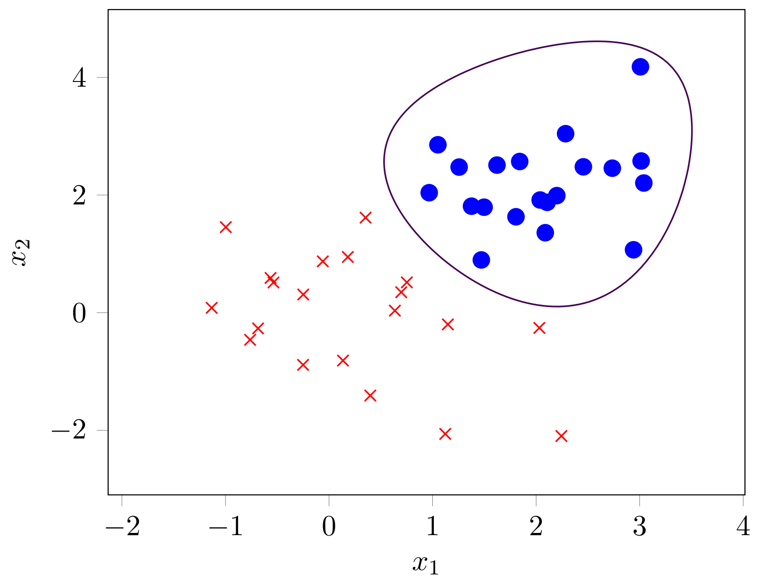

Figure 2: Decision boundary for non-linear logistic regression.

1.3 Logistic Regression — regularization

In this exercise we work with the file: `log_reg_ex.py`. First, modify the line which imports `DataSet` such that the import operation is performed from the file `dataset2_logreg`. The objective is to minimize the cost.

- Use $P = 8$. Find proper a GD-learning rate and a number of GD-iterations. You should obtain results similar to these on Figure 3 and cost around 7.5. This phenomenon is called over-fitting. The model is too complex and the neuron has also "learned the noise". The model contains 45 parameters in $\underline{w}$ but the DataSet were generated from a function containing roughly 5–10 parameters, so there is still space left for learning "too much". We would like to discourage complex model although it achieves better performance (lower cost) - this process is called **regularization**. We would like to have a simpler model but still flexible, i.e., keep high value of P (higher P allows for more complex decision boundaries when needed). This can be done by **weight decay**. We modify the cost function by adding the term $\lambda \|\underline{w}\|^2$ with some constant $\lambda \geq 0$. This introduces penalty for using parameters with higher magnitude, therefore limits our parameter space and makes models simpler. Higher value of λ penalize $\|\underline{w}\|^2$ more and result in simpler models.

- Change `mlcomm.nn.utils.lor_cost(w, y, x)` to `mlcomm.nn.utils.lor_cost(w, y, x, lbd)` and add the term $\lambda \|\underline{w}\|^2$ to the current cost function. I.e., the cost

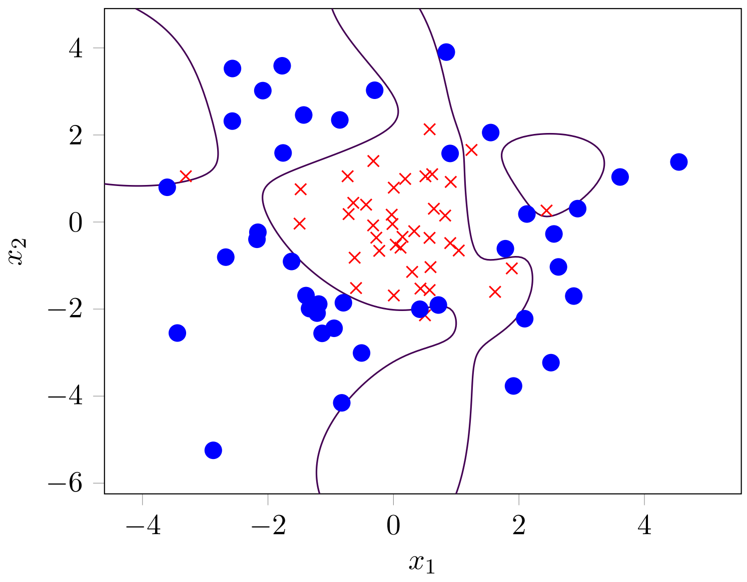

Figure 3: Over-fitting.

function is now:

$$\sum_{i=1}^{N} \left\{ -y^{(i)} \log(a^{(i)}) - (1-y^{(i)}) \log(1-a^{(i)}) \right\} + \lambda \|\underline{w}\|^2$$

- Change `mlcomm.nn.utils.lor_grad(w, y, x)` to `mlcomm.nn.utils.lor_grad(w, y, x, lbd)`. Derive the expression for the gradient from the new cost function and implement it.
- Run the exercise for $\lambda = 0$ (you can provide the parameter λ to the gradient function inside the gradient wrapper). You should obtain the same results. Run the exercise for $\lambda = 0.1$. You should obtain result similar to the one on Figure 4.

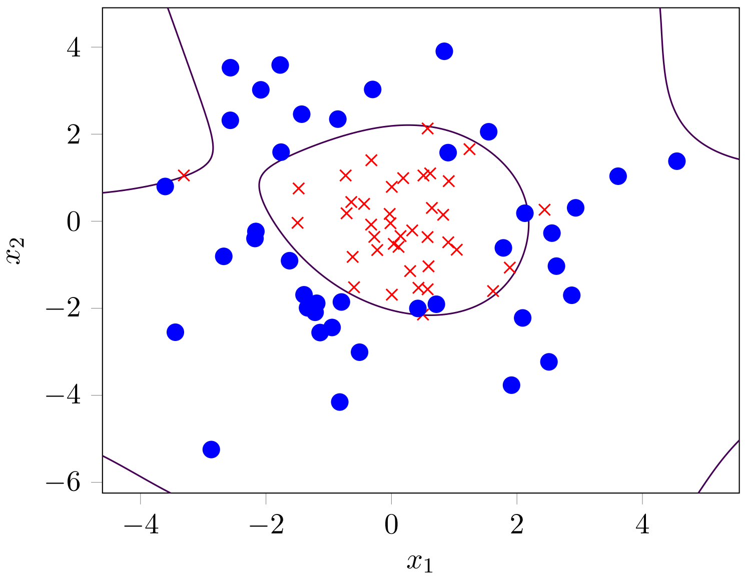

Figure 4: Regularization.