HANDS-ON AI I

Logistic Regression as a Door Opener to Deep Learning

Sohvi Luukkonen
Institute for Machine Learning

JOHANNES KEPLER
UNIVERSITY LINZ

Institute for
Machine Learning

Copyright Statement

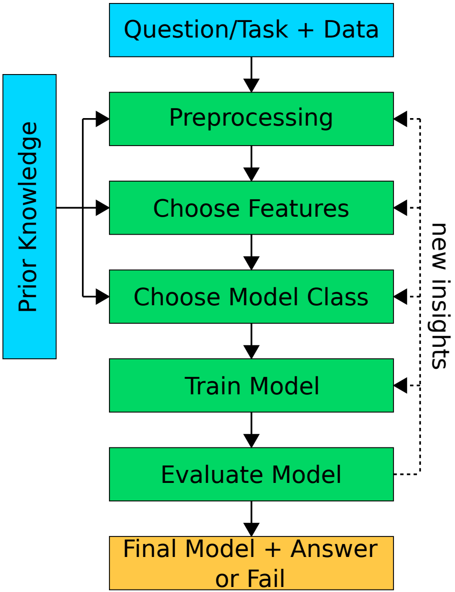
Content of Unit 4

- Short recap
- Linear regression
- Logistic regression
  - Loss functions
  - Optimization and Gradient Descent
- PyTorch as your first Deep Learning framework

From Logistic Regression to Deep Learning

- This lecture comprises an introduction to logistic regression and **PyTorch**, which is the deep learning tool you will use from now on.
- In the next lecture/exercise, you will see how coding up logistic regression in PyTorch looks like.
- And from there it is easy to expand the code towards neural networks.
- For those you are especially curious, the logistic regression PyTorch model is hidden in the function `minimize_ce` in the `04_utils.py` file.

Recap: Basic Data Analysis Workflow



Recap: Scoring Our Models: Loss Function

- Assume we have a model g, parameterized by $\boldsymbol{w}$.
- $g(\boldsymbol{x}; \boldsymbol{w})$ maps an input vector $\boldsymbol{x}$ to an output value $\hat{y}$.
- We want $\hat{y}$ to be as close as possible to the true target value y.
- We can use a **loss function**

$$L(y, g(\boldsymbol{x}; \boldsymbol{w})) = L(y, \hat{y})$$

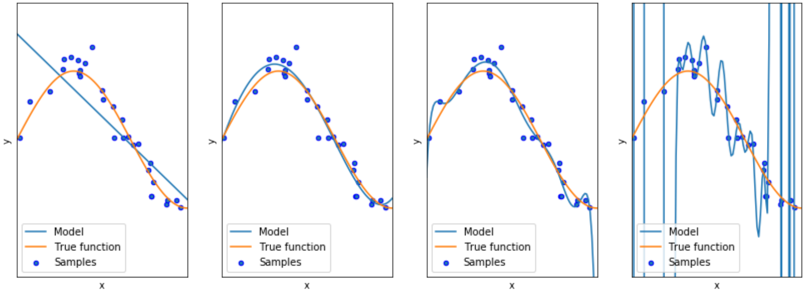
  to measure how close our prediction is to the true target for a given sample with $(\boldsymbol{x}, y)$.
- **The smaller the loss/cost, the better our prediction**.

Recap: Training and Test Data Sets

- Assume our data samples are **independently and identically distributed (i.i.d.)**[1]
- We can split our data set of n samples into **two non-overlapping subsets**:
  - **Training set**: a subset with l samples we perform ERM on (i.e., optimize parameters on)
  - **Test set**: a subset with m samples we use to estimate the risk (test data = approximation of future, unseen data)
- Our estimate R_E on the test set will show if we overfit to noise in the training set.

[1] i.i.d.: Each sample has the same probability distribution as the others, and all samples are mutually independent.

Recap: Underfitting and Overfitting



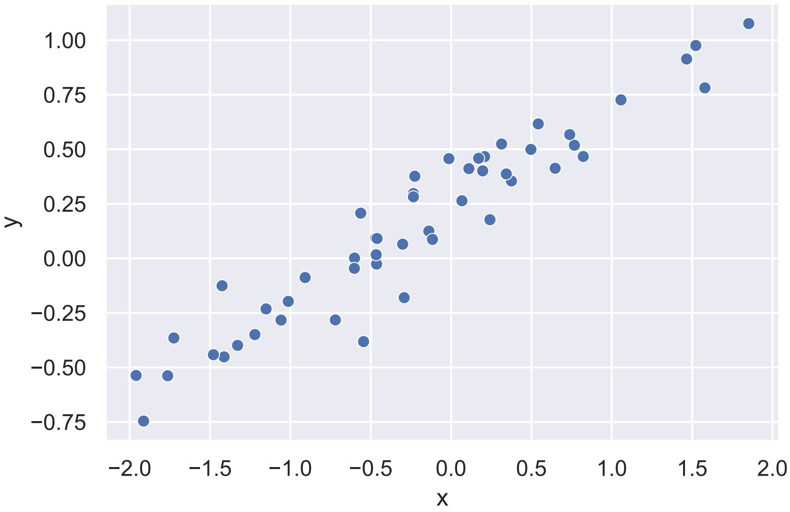
Linear Regression

- Linear Regression is one of the simplest machine learning algorithms.
- Linear Regression $\Rightarrow$ Logistic Regression $\Rightarrow$ Neural Networks $\Rightarrow$ Deep Learning

Formulas

- Given labeled data $(\boldsymbol{X}, \boldsymbol{y}) = ((\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n))$
  - $\boldsymbol{x}_i$: feature vector
  - y_i: corresponding label
  - n: number of samples (data set size)
- Find model $g(\boldsymbol{x}; \boldsymbol{w})$ such that $\forall i : g(\boldsymbol{x}_i; \boldsymbol{w}) \approx y_i$.
- **Regression** $\Leftrightarrow y_i \in \mathbb{R}$
- **Classification** $\Leftrightarrow y_i \in \{1, \ldots, K\}$

Example



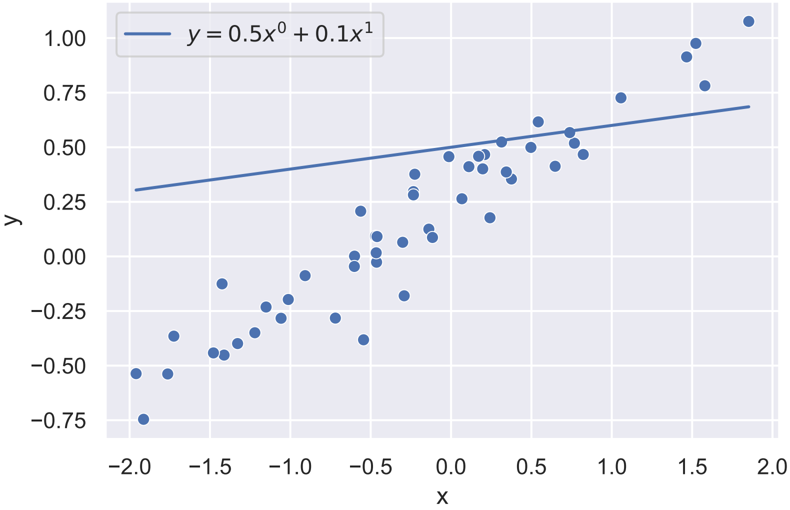
Linear Model

- Simplest approach: use something linear:

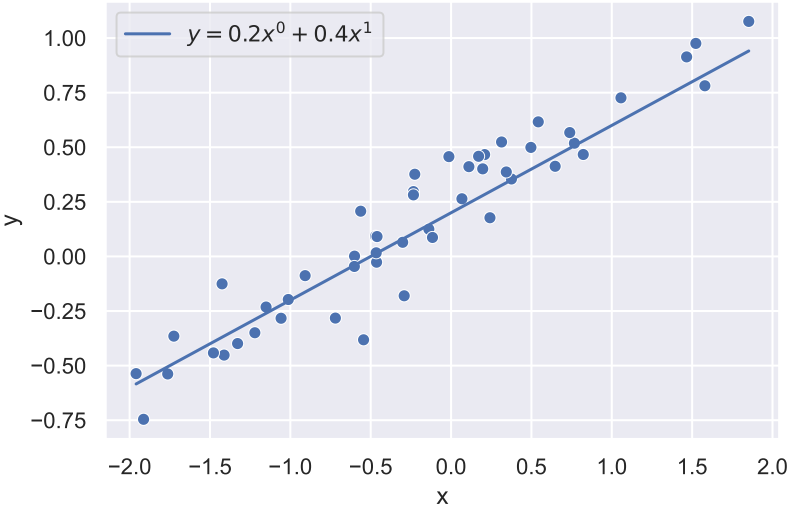
$$g(\boldsymbol{x}_i; \boldsymbol{w}) = g(\boldsymbol{x}_i; a, b) = a + b \cdot \boldsymbol{x}_i$$

- a and b are our model parameters.
- We must find some fitting a and b that properly model the data.

Linear Model: $a = 0.5, b = 0.1$



Linear Model: $a = 0.2, b = 0.4$



Finding Good Parameters

- Our linear model:

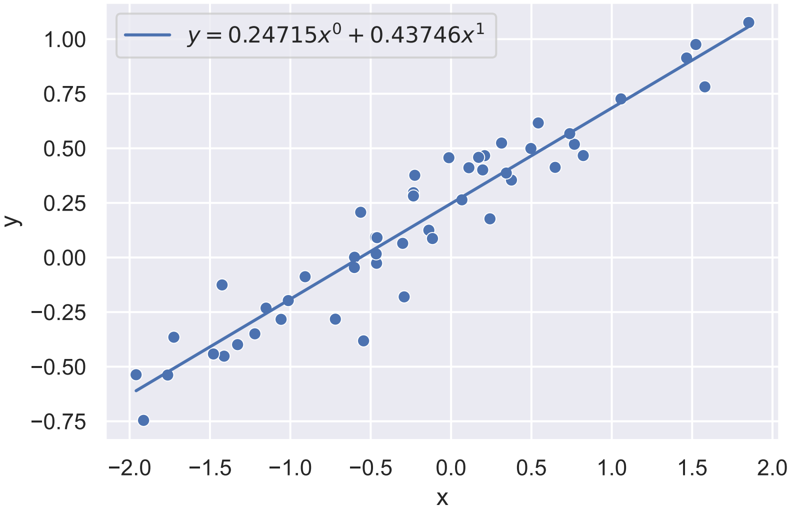
$$g(\boldsymbol{x}_i; \boldsymbol{w}) = g(\boldsymbol{x}_i; a, b) = a + b \cdot \boldsymbol{x}_i$$

- **Linear regression**: Fit this linear model to minimize some error/loss.
- Idea: **Minimize mean-squared error loss**:

$$L = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - g(\boldsymbol{x}_i; \boldsymbol{w})\right)^2$$

- With linear regression, finding the minimum loss can be done **analytically**, i.e., it essentially can be computed directly from the data set.

Linear Regression



Polynomials Are Linear Models

- Instead of using something linear:

$$g(\boldsymbol{x}_i; \boldsymbol{w}) = a + b \cdot \boldsymbol{x}_i$$

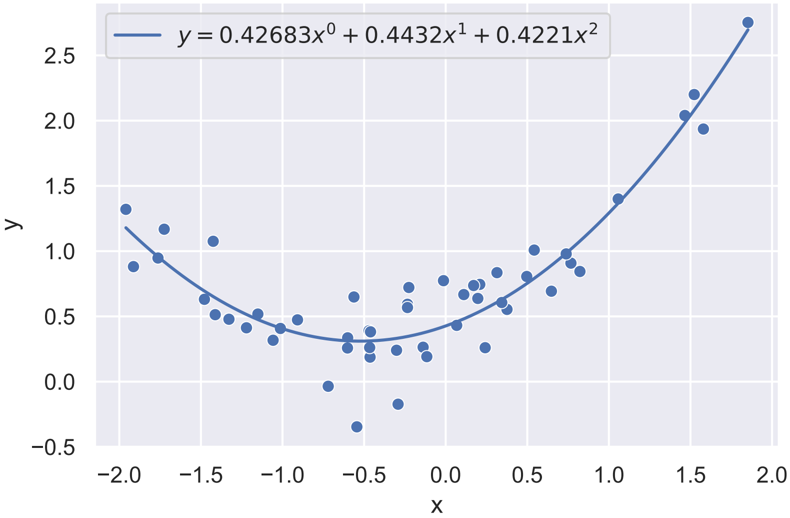
- we can also use any polynomial (linear in the coefficients):

$$g(\boldsymbol{x}_i; \boldsymbol{w}) = a + b \cdot \boldsymbol{x}_i + c \cdot \boldsymbol{x}_i^2 + d \cdot \boldsymbol{x}_i^3 + \ldots$$

- Main idea to find best parameters is the same: **minimize mean-squared error loss**:

$$L = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - g(\boldsymbol{x}_i; \boldsymbol{w})\right)^2$$

Polynomials Are Linear Models

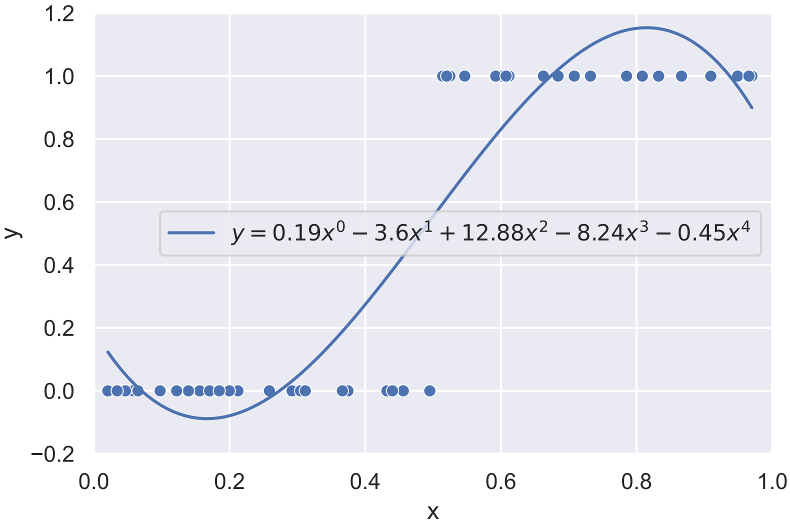


Problems With Linear Regression

- Given: n datapoints $\boldsymbol{x}_i$ with labels $y_i \in \{0, 1\}$
- Task: find $g(\boldsymbol{x}; \boldsymbol{w})$ such that $g(\boldsymbol{x}_i; \boldsymbol{w}) = y_i$
  $\Rightarrow$ **Classification task**
- First (bad) idea: fit a linear regression line
- Then, get classes based on some threshold:

$$y_i = \begin{cases} 0 & g_{\text{LinReg}}(\boldsymbol{x}_i; \boldsymbol{w}) < 0.5 \\ 1 & g_{\text{LinReg}}(\boldsymbol{x}_i; \boldsymbol{w}) \geq 0.5 \end{cases}$$

Problems With Linear Regression



Logistic Regression

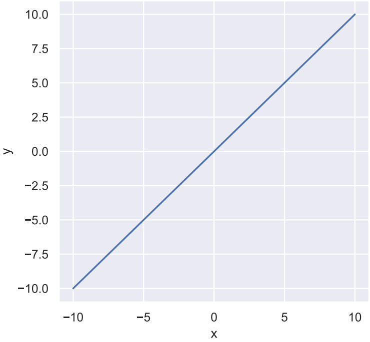
- Problem: The relationship between features $\boldsymbol{x}_i$ and labels y_i is **not linear**.
- Idea: Use something **non-linear** instead, e.g., apply the **logistic function** (also known as **sigmoid function**):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

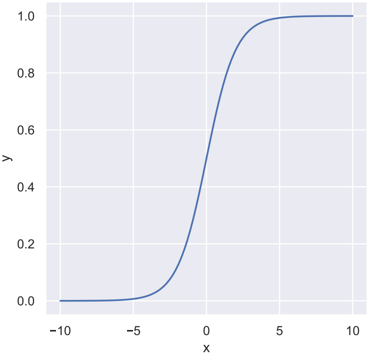
  where z is a linear function of features (in our case, this will be the "raw"/linear model output).
- Exemplary logistic model ("wrapped" linear model):

$$g(\boldsymbol{x}_i; \boldsymbol{w}) = \sigma(a + b \cdot \boldsymbol{x}_i)$$
$$\boldsymbol{w} = \{a, b\}$$

Logistic/Sigmoid Function



$\Rightarrow$



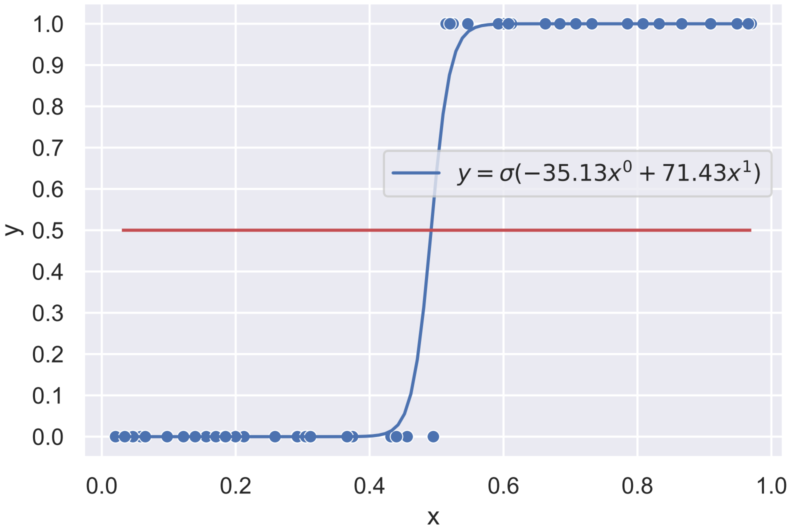
- Values are squashed to the range $[0, 1]$.
- Can be interpreted as probabilities.

Logistic Regression

- Given: n datapoints $\boldsymbol{x}_i$ with labels $y_i \in \{0, 1\}$
- Task: find $g(\boldsymbol{x}; \boldsymbol{w})$ such that $g(\boldsymbol{x}_i; \boldsymbol{w}) = y_i$
  $\Rightarrow$ **Classification task**
- New (better) idea: apply logistic regression
- Then, get classes based on some threshold:

$$y_i = \begin{cases} 0 & g_{\text{LogReg}}(\boldsymbol{x}_i; \boldsymbol{w}) < 0.5 \\ 1 & g_{\text{LogReg}}(\boldsymbol{x}_i; \boldsymbol{w}) \geq 0.5 \end{cases}$$

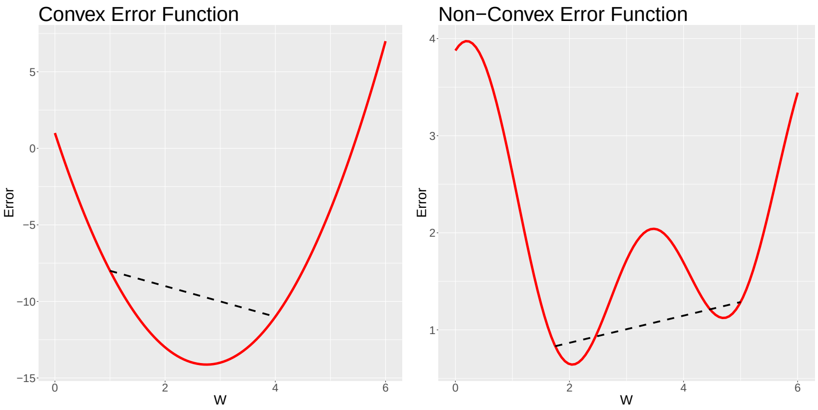
Logistic Regression



Logistic Regression or Logistic Classification?

- Logistic regression is emphatically not a classification algorithm on its own.
- It is only a **classification algorithm in combination with a decision rule** (e.g., based on some threshold) that makes the predicted probabilities of the outcome.
- Logistic regression is a **regression model** because it estimates the **probability of class membership** (output y is a real numeric value, i.e., $y \in [0, 1] \in \mathbb{R}$).

Logistic Regression Is a Convex Problem



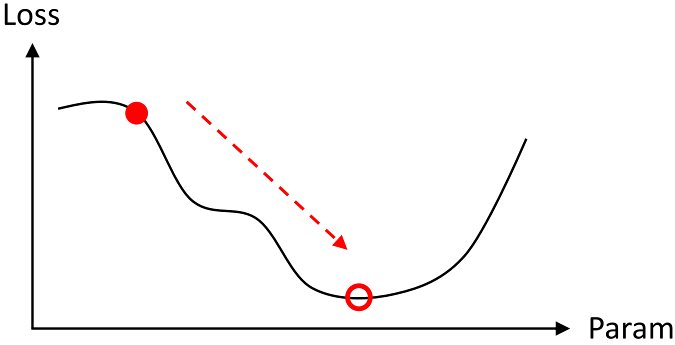
- A convex function has only one minimum (local minimum = global minimum).
- Logistic regression is a convex problem.

Logistic Regression Has No Closed Form Solution

- Despite being a convex problem, logistic regression has **no closed-form solution**.
- No closed-form solution: minimum of the loss cannot be calculated directly (no analytical solution).
- **Iterative methods** have to be used for minimizing the loss.
- One prominent example is **gradient descent**.

Gradient Descent

- Given: a function $f(x)$
- Task: find x that maximizes (or minimizes) $f(x)$
- Idea: start at some value x_0, and take a small step η in the direction in which the function decreases strongest



Gradient Descent in Logistic Regression

- The minimization of the loss function $L(.;\theta)$ can be done by gradient descent:

$$\theta_{n+1} = \theta_n - \eta \frac{\partial L}{\partial \theta}$$

  where η is the learning rate and θ is the parameter or set of parameters to be optimized.

- In the case

$$g(\boldsymbol{x}_i; \boldsymbol{w}) = \sigma(a + b \cdot \boldsymbol{x}_i)$$

  the set of parameters is $\theta = \boldsymbol{w} = \{a, b\}$.

Softmax

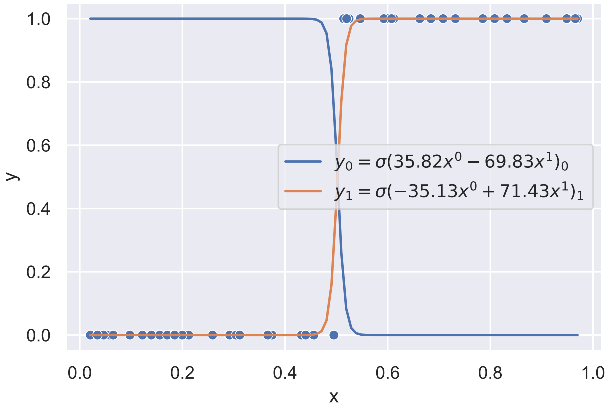
- Generalization of the sigmoid function.
- Suitable for **multi-class** classification.
- For K classes with $y \in \{1, \ldots, K\}$ the probability of $\boldsymbol{x}$ belonging to class i is:

$$p(y = i \mid \boldsymbol{x}) = \sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

  where $\boldsymbol{z} = (z_1, \ldots, z_K)$ is the vector of “raw” model outputs, i.e., there is an output for each class (see example later).
- While not necessary, as a regular sigmoid function would suffice, this also works for binary classification, i.e., $K = 2$.

Softmax

- Same example as earlier with $K = 2$ classes:



- Here, $y_0 = p(y = 0 \mid \boldsymbol{x})$ and $y_1 = p(y = 1 \mid \boldsymbol{x})$, and the final model output would be (y_0, y_1).

Softmax

- Another example:
  - Image you have a data set with $K = 10$ classes.
  - For $p(y = 0 \mid \boldsymbol{x})$, your model should ideally output $(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$.
  - For $p(y = 1 \mid \boldsymbol{x})$, your model should ideally output $(0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$.
  - But also the output $(0.01, 0.7, 0.02, 0.02, 0.05, 0.1, 0.01, 0.02, 0.04, 0.03)$ is already pretty good.
- To achieve outputs like that, we use the so-called **cross-entropy loss** for classification (with special case **binary cross-entropy** for binary classification).

Summary

- **Classification**: target value is class label
- **Regression**: target value is numerical value
- Linear Regression $\Rightarrow$ Logistic Regression $\Rightarrow$ Neural Networks $\Rightarrow$ Deep Learning
- **Mean-squared error** as loss for regression.
- **Cross-entropy error** as loss for classification.

Summary

- We introduced linear regression as one of the simplest machine learning algorithms.
- We broke linear regression with a simple classification task.
- We introduced logistic regression to fix our model.
  - kNN and Random Forest approaches would also have been able to solve such simple problems.
- We extended the model to multiple class outputs.
- We can now solve a huge variety of tasks.
- Next time, we will extend further and build our first neural network.

PyTorch

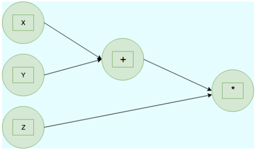
Introduction

An **open source deep learning platform for Python** that is very well suited for experimental approach and prototyping as well as production:

- Easy to build big computational graphs.
- **Automatic computation of gradients** for learning.
- Smoothly switch between CPU and GPU.
- High execution efficiency, written in C and CUDA.

Computational Graphs

- A computational graph is a way to represent a math function in the language of graph theory.
- **Graph theory**: nodes are connected by edges, and everything in the graph is either a node or an edge.
- **Nodes** are either input values or functions for combining values.
- **Edges** receive their weights as the data flows through the graph.
- Consider the relatively simple expression: $f(x, y, z) = (x + y) \cdot z$



Tensors, Modules

PyTorch can be thought of as consisting of three levels of abstraction:

- **Tensors** represent tensors of any order (e.g., a scalar is a 0th order tensor), technically equivalent to numpy arrays, with some additional features like the ability to put it on the GPU.
- **Computational graphs**: A PyTorch tensor represents a node in a computational graph. If `x` is a tensor that has `x.requires_grad=True`, then `x.grad` is another tensor holding the gradient of `x` with respect to some scalar value.
- **Modules**: e.g., modules for neural networks (layers); allow for composition (a module can be composed of other modules).

Autograd

- In computational graphs, the **provenance of tensors is tracked**, i.e., they know how they are constructed.
- The computation of the gradient comes therefore for free: automatic computation of the gradient is called **"autograd"**.
- No need to compute gradients ourselves!