# Logistische Regression - Einführung

August 17, 2019

## 1 Logistische Regression

### 1.1 Theorie: Von der `Linearen Regression` zur `Logistischen Regression`

#### 1.1.1 Eine "Erweiterung" der `Linearen Regression`

Wir haben uns bislang mit der `Linearen Regression` vor dem Hintergrund beschäftigt, dass wir den Prozess in einem Neuron modellieren wollen. Dementsprechend haben wir die Grundlagen für ein **lineares Neuron** gelegt. In einem `Neuronalen Netz` wollen wir viele solcher Neuronen in Schichten hintereinanderschalten ("stacking"), d.h. die Ausgabe-Werte verschiedener Neuronen werden an die Neuronen der nächsten Schicht übergeben, von ihnen weiterverarbeitet und dann an die Neuronen der nächsten Schicht übergeben usw. Dafür sind lineare Neuronen nicht geeignet.

Daher werden wir zur `Logistischen Regression` übergehen, die geeigneter ist, um zu beschreiben, was in den einzelnen Neuronen passiert. Die `Logistische Regression` läuft sehr ähnlich wie die `Lineare Regression` ab, aber anders als die `Lineare Regression` löst man damit **Klassifizierungsprobleme**. Bei der `Linearen Regression` haben wir stetige Werte vorhergesagt, mit der `Logistischen Regression` ist der Output diskret, d.h. man kann die möglichen Ergebnisse zählen. Wir können also anhand gegebener Merkmale beurteilen, in welche vorgegebene Klasse eine Beobachtung fällt und somit "Ja/Nein" - Entscheidungen treffen. Zum Beispiel können wir die `Logistische Regression` dazu einsetzen, um Emails als "Spam" oder "Kein Spam" zu klassifizieren oder Tiere auf einem Foto zu identifizieren.

#### 1.1.2 Faustformel: "Logistische Regression = Lineare Regression + Aktivierungsfunktion"

Im Wesentlichen unterscheidet sich die `Logistische Regression` von der der `Linearen Regression`, indem man den Datensatz durch eine komplexere Funktion zu modellieren versucht: Man setzt dazu die bisherige lineare Funktion `y = a * x + b` in eine nicht-lineare Funktion `S` ein, sodass wir nunmehr die Daten durch diese neue Funktion `S(y) = S(a * x + b)` beschreiben. Wir müssen weiterhin die Parameter a und b optimieren!

Die Idee hinter dieser nicht-linearen Funktion S ist, dass man sie als **Aktivierungsfunktion** benutzt, um über die Werte aus der linearen Funktion zu entscheiden: Wenn der lineare Term negative Werte liefert, dann würden wir die Klassifizierung ablehnen, sonst annehmen. Für diese nicht-lineare Funktion strebt man also eine Art Stufenfunktion an, die alle Werte unterhalb einer bestimmten Grenze auf einen konstanten Wert setzt (Ablehnung) und alle Werte oberhalb dieser Grenze auf einen anderen konstanten Wert (Zustimmung). Zum Beispiel kann man sich eine Funktion vorstellen, die, wenn man negative Zahlen einsetzt, 0 ist und bei positiven Zahlen den Wert 1 annimmt.

Mit einer solchen Funktion lässt sich aber nicht gut rechnen, weil sie nicht stetig ist; insbesondere müssen wir berücksichtigen, dass wir sie für das Gradientenverfahren noch in die Kostenfunktion einsetzen und ableiten müssen.

Deswegen wählen wir stattdessen eine ähnliche Funktion, die für uns angenehmere Eigenschaften besitzt, also insbesondere eine gut darstellbare Ableitung besitzt.

### 1.1.3 Die Sigmoid-Funktion

Die sogenannte Sigmoid-Funktion bildet alle Zahlen auf den Bereich von 0 bis 1 ab.

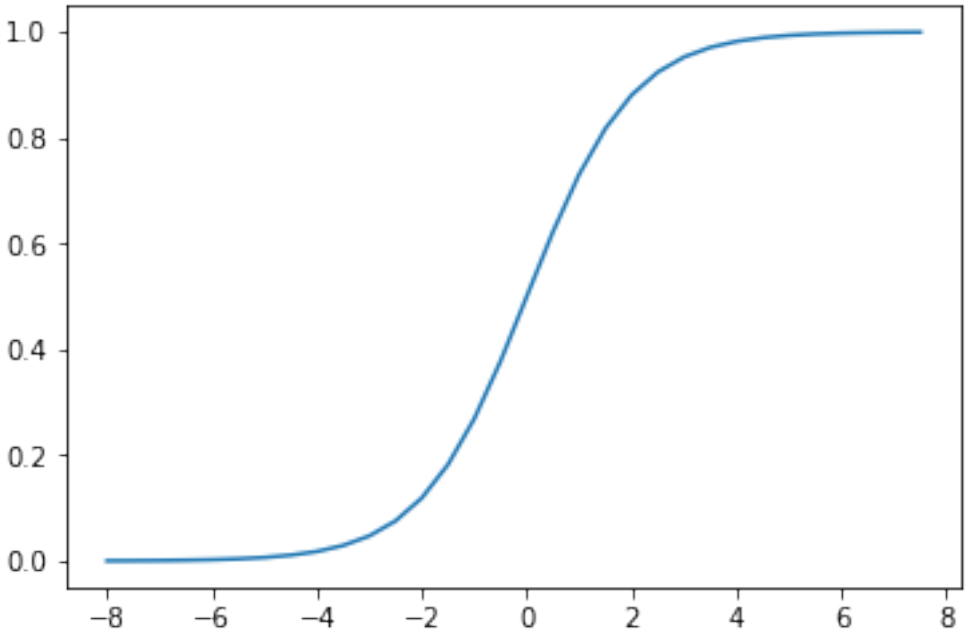
$$S(x) = \frac{1}{1 + e^{-x}}$$

e bezeichnet die Eulersche Zahl, eine Naturkonstante, die etwa 2,72 beträgt.

Du kannst dir den Graphen als S - Kurve vorstellen. Zur Verabschaulichung plotten wir sie an dieser Stelle:

[1]:
```
# die NumPy - Bibliothek für numerische Berechnungen importieren
import numpy as np

# das matplotlib.pyplot - Paket für Plots importieren
import matplotlib.pyplot as plt
```

[2]:
```
# Sigmoid-Funktion definieren
def S(x):
    return 1/(1 + np.exp(-x))

# NumPy - Array mit Zahlen zwischen -8 und 8 erstellen
x = np.arange(-8, 8, 0.5)

# auf jedes Element des Arrays die Sigmoid - Funktion anwenden
y = S(x)

# Plotten
plt.plot(x, y)

# Plot anzeigen
plt.show()
```




Du siehst, dass größere positive Zahlen ganz nah an 1 abgebildet werden und ganz kleinere negative Zahlen ganz nah an 0. Doch nimmt die Sigmoid-Funktion 0 oder 1 an keiner Stelle an!

Ein Vorteil der Sigmoid-Funktion im Vergleich zu der oben erwähnte Stufenfunktion besteht darin, dass sie besser abzuleiten ist.

%

$$S'(x) \underset{\text{Kettenregel}}{=} (-e^{-x}) \cdot -\frac{1}{(1+e^{-x})^2} \underset{0 \text{ ergänzen}}{=} \frac{-1+1+e^{-x}}{(1+e^{-x})^2}$$

$$= \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})} \cdot \frac{1}{(1+e^{-x})} = S(x)(1-S(x))$$

**Die Ableitung der Sigmoidfunktion lässt sich also durch Sigmoidfunktionen darstellen und sich daher in einer angenehmen Form angeben!**

### 1.1.4 Wie können wir die Sigmoid - Funktion interpretieren?

Angenommen, wir wollen ein binäres Klassifikationsproblem lösen, uns interessiert also nur, in welche von zwei Klassen unsere Daten fallen (z.B. “Spam” vs “Kein Spam” oder “Tumor” vs “Kein Tumor”). Dann benutzen wir als y - Werte nur die Werte 0 (z.B. “Kein Spam” oder “Kein Tumor”) und 1 (für die komplementäre Klasse, z.B. “Spam” oder “Tumor”).

Wenn wir dann mit unserem Modell aus der `Logistischen Regression` neue Daten klassifizieren wollen, erhalten wir wegen der Sigmoid - Funktion dazu eine Zahl zwischen 0 und 1. Diese Zahl können wir als **Wahrscheinlichkeit** bezüglich der binären Klassifikation zu interpretieren. Also würde ein Wert von 0.7 etwa aussagen, dass bei der Mail bzw. dem Patienten mit 70% - iger Wahrscheinlichkeit “Spam” bzw. ein “Tumor” vorliegt.

Insbesondere fungiert der Wert von 0.5 als Entscheidungsgrenze.

### 1.1.5 Optional: Kostenfunktion

Auch die Kostenfunktion muss im Vergleich zur `Linearen Regression` angepasst werden:

$h_\theta(x^{(i)})$ : Ausgabe des Modells für den Punkt $x^{(i)}$; nimmt nur Werte zwischen 0 und 1 an

$\theta$: bezeichnet einen "Container" für alle Parameter, von denen h noch abhängt, in unserem Fall: $\theta = (a, b)$

$y^{(i)}$ : der tatsächliche y - Wert, also 0 oder 1

Damit formulieren wir nun die neue Kostenfunktion, indem wir die Kosten zu jedem einzelnen Punkt aufsummieren und dann durch die Anzahl m der Punkte mitteln:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

Die wesentliche Veränderung besteht nun darin, wie wir die Kosten für jeden einzelnen Punkt berechnen:

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)), \text{ wenn y = 1} \\ -log(1 - h_\theta(x)), \text{ wenn y = 0} \end{cases}$$

Beachte, dass der Logarithmus dann gleich 0 ist (in diesem Fall also auch die Kosten gleich 0 sind), wenn man 1 in die Logarithmus - Funktion einsetzt. Je näher die eingesetzte Zahl (aus dem Intervall von 0 bis 1) an 0 liegt, desto größer wird der Wert des negativen Logarithmus, desto größer sind also unsere Kosten und desto schlechter ist unsere Schätzung.

Wenn man bei y = 0 gerade 0 vorhersagen will, sorgt ein großer Wert von h dafür, dass `(1 - h)` nahe bei 0 liegt und dementsprechend der negative Logarithmus einen sehr großen Wert liefert (hohe Kosten). Ist in diesem Fall h hingegen nahe bei 0, dann liegt `(1 - h)` nahe bei 1 und der Logarithmus davon wird einen Wert nahe 0 annehmen (geringe Kosten).

Eingesetzt ergibt sich als geschlossene Form für die Kostenfunktion, wenn wir die Fallunterscheidung auflösen:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\theta(x^{(i)}))$$

Denn entweder für jedes i ist entweder y gleich 1 und dann (1 - y) gleich 0 oder umgekehrt, anhängig von y fällt also der andere (unpassende) Teil aus der Summe weg.

Nun müssen wir noch die partielle Ableitungen für die Kostenfunktion J bestimmen, also J nach a und b ableiten. Davor unternehmen wir allerdings erst einige Vorbereitungen, damit das Ableiten leichter fällt.

In unserem Fall gilt:

$$h_\theta(x) = h_{(a,b)}(x) = S(a \cdot x + b) = \frac{1}{1 + e^{-(a \cdot x + b)}}$$

Damit vereinfachen sich die Ausdrücke in der Kostenfunktion wie folgt:

$$(*) : log(h_\theta(x^{(i)})) = log(\frac{1}{1 + e^{-(a \cdot x^{(i)} + b)}}) = log((1 + e^{-(a \cdot x^{(i)} + b)})^{-1}) = -log(1 + e^{-(a \cdot x^{(i)} + b)})$$

$$(**) : log(1 - h_\theta(x^{(i)})) = log(\frac{1 + e^{-(a \cdot x^{(i)} + b)}}{1 + e^{-(a \cdot x^{(i)} + b)}} - \frac{1}{1 + e^{-(a \cdot x^{(i)} + b)}}) = log(\frac{e^{-(a \cdot x^{(i)} + b)}}{1 + e^{-(a \cdot x^{(i)} + b)}}) = log(e^{-(a \cdot x^{(i)} + b)}) - log(1 + e^{-(}$$

$$= -(a \cdot x^{(i)} + b) - log(1 + e^{-(a \cdot x^{(i)} + b)})$$

Einsetzen liefert nun:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \cdot log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \cdot log(1 - h_\theta(x^{(i)})) \underset{(*)}{=} -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \cdot (-log(1 + e^{-(a \cdot x^{(i)} + b)})) + (1 - y^{(i)}) \cdot log(1 - h$$

$$\underset{(**)}{=} -\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \cdot (-log(1 + e^{-(a \cdot x^{(i)} + b)})) + (1 - y^{(i)}) \cdot (-(a \cdot x^{(i)} + b) - log(1 + e^{-(a \cdot x^{(i)} + b)}))$$

$$= -\frac{1}{m} \sum_{i=1}^{m} (-(a \cdot x^{(i)} + b) - log(1 + e^{-(a \cdot x^{(i)} + b)})) + y^{(i)} \cdot (a \cdot x^{(i)} + b)$$

$$= -\frac{1}{m} \sum_{i=1}^{m} -(log(e^{(a \cdot x^{(i)} + b)}) + log(1 + e^{-(a \cdot x^{(i)} + b)})) + y^{(i)} \cdot (a \cdot x^{(i)} + b)$$

$$= -\frac{1}{m} \sum_{i=1}^{m} -log(1 + e^{(a \cdot x^{(i)} + b)}) + y^{(i)} \cdot (a \cdot x^{(i)} + b)$$

Von diesem Ausdruck können wir nun relativ schnell die partiellen Ableitungen berechnen:

$$\frac{\partial J(\theta)}{\partial a} = -\frac{1}{m} \sum_{i=1}^{m} -\frac{x^{(i)}}{1 + e^{(a \cdot x^{(i)} + b)}} + y^{(i)} \cdot x^{(i)} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \cdot S(a \cdot x^{(i)} + b) - y^{(i)} \cdot x^{(i)} = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} \cdot (S(a \cdot x^{(i)} + b) - y^{(i)})$$

$$\frac{\partial J_\theta}{\partial b} = -\frac{1}{m} \sum_{i=1}^{m} -\frac{1}{1 + e^{(a \cdot x^{(i)} + b)}} + y^{(i)} = \frac{1}{m} \sum_{i=1}^{m} S(a \cdot x^{(i)} + b) - y^{(i)}$$

Puh, geschafft!!! :-)

## 1.2 Implementierung einer `Logistischen Regression`: Volljährigkeit vorhersagen

Um die Logistische Regression besser zu verstehen, werden wir uns nun ein leichtes Klassifizierungsproblem anschauen. Wir wollen anhand des Alters "Volljährigkeit / Keine Volljährigkeit" klassifizieren. Unsere x - Daten sind Jahreszahlen und die y - Werte beschreiben entweder 0 ("Nicht volljährig") oder 1 ("volljährig").

Bedenke, dass der Algorithmus kein Konzept wie Volljähirgkeit kennt, d.h. er muss aus unserem Datensatz herausfinden, wo die Grenze zu ziehen ist.

### 1.2.1 Datensatz generieren

[4]:
```
# 1. Spalte: Alter
# 2. Spalte: volljährig (1) / Nicht volljährig (0)

points = np.array([
    [20, 1],
    [17, 0],
    [15, 0],
    [10, 0],
    [30, 1],
    [40, 1],
    [35, 1],
    [13, 0],
    [5, 0],
    [18, 1],
    [25, 1],
    [8, 0]
])
```

### 1.2.2 Funktionen für `Logistische Regression` anpassen

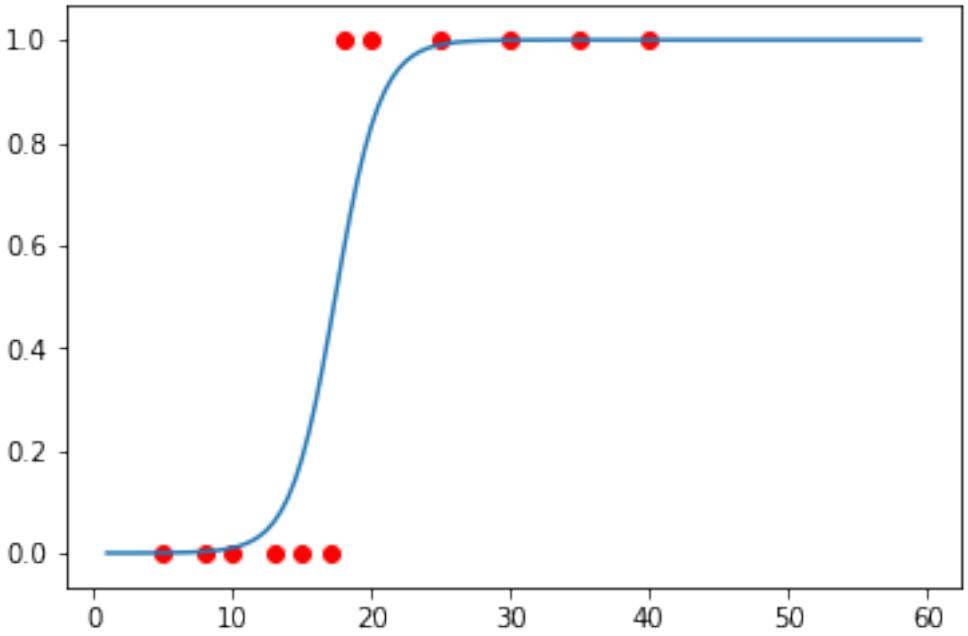
[9]:
```
# nicht-lineare Funktion mithilfe der Sigmoid - Funktion
def f(a, b, x):
    return S(a * x + b) # statt: return a * x + b
```

[10]:
```
# Kostenfunktion
def J(a, b, x, y):
    return -np.mean(y * np.log(f(a, b, x)) + (1 - y) * np.log(1 - f(a, b, x)))
    # statt np.mean((y - a * x) ** 2)
```

[11]:
```
def J_ableitung_a(a, b, x, y):
    return np.mean(x * (S(a * x + b) - y))

def J_ableitung_b(a, b, x, y):
    return np.mean(S(a * x + b) - y)
```

### 1.2.3 Ab hier: denselben Code wie bei der `Linearen Regression` verwenden :-)

[19]:
```
# Mit Gradientenverfahren Parameter a und b in Kostenfunktion optimieren

# Lernrate heraufsetzen
lr = 0.05

# Startwerte für zu optimierende Parameter
a = 1
b = 1

# Anzahl der Schleifendurchläufe heraufsetzen
for i in range(0, 10000):

    # Ableitungen bestimmen
    da = J_ableitung_a(a, b, points[:, 0], points[:, 1])
    db = J_ableitung_b(a, b, points[:, 0], points[:, 1])

    # Punkte aktualisieren
    a = a - lr * da
    b = b - lr * db

    # Kosten berechnen
    cost = J(a, b, points[:, 0], points[:, 1])


# Plotten

# Logistische Regressions - Kurve einzeichnen
xs = np.arange(1, 60, 0.5)
ys = f(a, b, xs)
plt.plot(xs, ys)

# Datenpunkte eintragen
plt.scatter(points[:, 0], points[:, 1], c="r")

plt.show()
```




### 1.2.4 Vorhersagen treffen

[13]:
```
# eine Person
print(f(a, b, 28))
```

```
0.9985715546285736
```

[14]:
```
# mehrere Personen
print(f(a, b, np.array([16, 22])))
```

```
[0.29247925 0.94444285]
```

[20]:
```
# mehrere, zufällige Personen

# Array mit zehn Zufallszahlen zwischen 0 und 39
ages = np.random.randint(0, 40, 10)

for i in ages:
    if f(a, b, i) > 0.5:
        print(str(i) + " -> Volljährig")
    else:
        print(str(i) + " -> Nicht volljährig!")
```

```
4 -> Nicht volljährig!
34 -> Volljährig
17 -> Nicht volljährig!
```

```
11 -> Nicht volljährig!
0 -> Nicht volljährig!
21 -> Volljährig
29 -> Volljährig
36 -> Volljährig
32 -> Volljährig
25 -> Volljährig
```