Machine Learning: Logistische Regression

Präsentation: Vorname, Nachname

Lizenz:

deep.TEACHING

www.deep-teaching.org

Wiederholung

Was ist der Unterschied zwichen Regression und Klassifikation?

Welche 3 Komponenten benötigt der Machine Learning-Algorithmus einer Linearen Regression?

Warum ist Skalierung wichtig?

- Wie wird skaliert?
- Was wird skaliert?

Was ist der Unterschied zwischen Supervised und Unsupervised Machine Learning?

Regression und Klassifikation

Regression: Bestimmung kontinuierlicher Zielwerte für einen Datensatz.

Beispiel: ein Haus wurde **1970** erbaut, hat eine Fläche von **350 m²** und wurde **nicht renoviert**.

$\rightarrow$ Wie hoch ist der übliche **Marktpreis**?

Klassifikation: Einordnung eines Datensatzes in diskrete Klassen.

Beispiel: Ein Haus wurde **1970** erbaut, hat eine Fläche von **350 m²** und kostet **250000 €**.

$\rightarrow$ Wurde das Haus renoviert, **ja** oder **nein**?

Fishers Iris-Daten

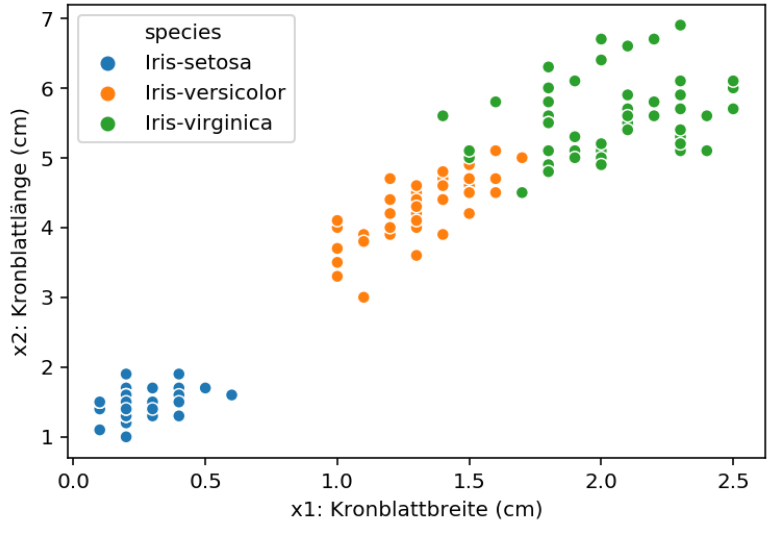
In [3]:
```
df = Iris(verbose=False).dataframe()
```

In [4]:
```
df['species'].value_counts() # Schwertlilien
```

Out[4]:
```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: species, dtype: int64
```

Plot der Daten

In [5]:
```
sns.scatterplot(data=df, x='petal_width', y='petal_length', hue='species')
plt.xlabel('x1: Kronblattbreite (cm)'); plt.ylabel('x2: Kronblattlänge (cm)');
```



Klassifikationsproblem

Unterscheidung von **Iris-versicolor** und **Iris-virginica** anhand der der zwei Features Kronblattbreite (**x1**) und Kronblattlänge (**x2**).

In [6]:
```
df_reduced = df.query('species == "Iris-versicolor" | species == "Iris-virginic
a"')
df_reduced.head()
```

Out[6]:

	sepal_length	sepal_width	petal_length	petal_width	species
50	7.0	3.2	4.7	1.4	Iris-versicolor
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
53	5.5	2.3	4.0	1.3	Iris-versicolor
54	6.5	2.8	4.6	1.5	Iris-versicolor

In [7]:
```
df_reduced['species'].value_counts()
```

Out[7]:
```
Iris-versicolor    50
Iris-virginica     50
Name: species, dtype: int64
```

X- und Y-Arrays

In [8]:
```
X = df_reduced[['petal_width', 'petal_length']].values
Y = df_reduced['species'].replace({'Iris-versicolor': 0, 'Iris-virginica': 1}).v
alues
```

In [9]:
```
X[:5]  # x1, x2 pairs
```

Out[9]:
```
array([[1.4, 4.7],
       [1.5, 4.5],
       [1.5, 4.9],
       [1.3, 4. ],
       [1.5, 4.6]])
```

In [10]:
```
Y[:5]
```

Out[10]:
```
array([0, 0, 0, 0, 0])
```

In [11]:
```
X.shape, Y.shape
```

Out[11]:
```
((100, 2), (100,))
```

Skalierung

In [12]:
```
scaler = StandardScaler()  # import aus Scikit-Learn
X_scaled = scaler.fit_transform(X)  # x1 und x2 werden separat skaliert
```

In [13]:
```
X_scaled[:5]
```

Out[13]:
```
array([[-0.65303909, -0.25077906],
       [-0.41643072, -0.49425387],
       [-0.41643072, -0.00730424],
       [-0.88964745, -1.10294091],
       [-0.41643072, -0.37251647]])
```

In [14]:
```
X_scaled.shape
```

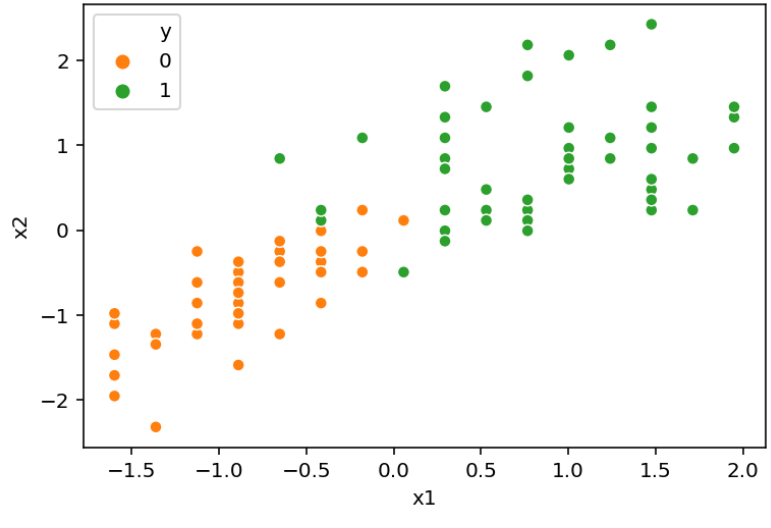
Out[14]:
```
(100, 2)
```

Plot der reduzierten und skalierten Daten

Wie gut kann eine Klassifikation anhand der Eingabe x_1, x_2 sein?

Wäre die Klassifikation mit nur einer Eingabevariable x_1 oder x_2 ähnlich gut?

In [16]:
```
df_scaled = pd.DataFrame(X_scaled, columns=['x1', 'x2'])
df_scaled['y'] = Y

sns.scatterplot(data=df_scaled, x='x1', y='x2', hue='y', palette=cp[1:3]);
```



Logistische Regression

Ein Datensatz gehört entweder zur Klasse 0 oder zur Klasse 1.

Die Hypothese der Logistischen Regression liefert für die Eingabe h(x1, x2) einen Wahrscheinlichkeitswert im Wertebereich $]0, 1[$ als Ausgabe. Je größer der Wert ist, desto wahrscheinlicher ist die Zugehörigkeit zu Klasse 1.

Logistische Funktion

Um die Ausgabe im Wertebereich $]0, 1[$ zu garantieren, wird die Logistische Funktion benötigt.

Die Logistische Funktion is ein Spezialfall der Sigmoid-Funktionen (oft als Synonym verwendet) und ist wie folgt definiert.

$$sigmoid(t) = \frac{1}{1 + e^{-t}}$$

In [18]:
```
sigmoid(2) # Implementierung in der Übung
```

Out[18]:
```
0.8807970779778823
```

In [19]:
```
sigmoid(2), sigmoid(4), sigmoid(6)
```

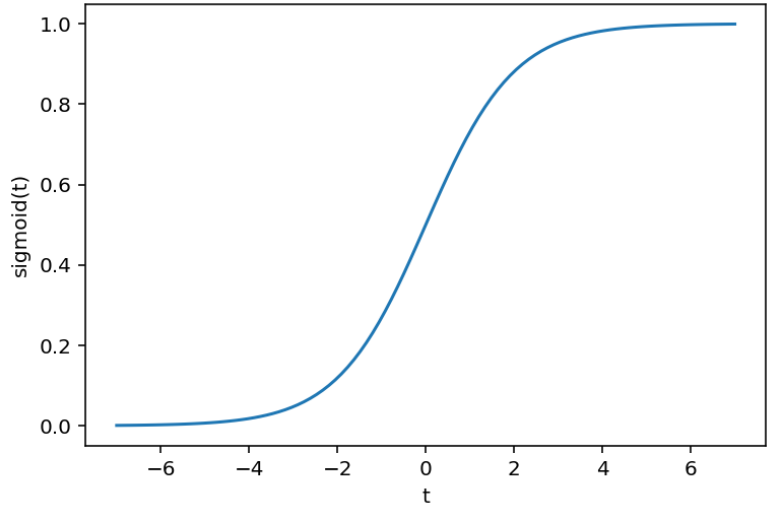
Out[19]:
```
(0.8807970779778823, 0.9820137900379085, 0.9975273768433653)
```

In [20]:
```
sigmoid(0), sigmoid(-2), sigmoid(-4), sigmoid(-6)
```

Out[20]:
```
(0.5, 0.11920292202211755, 0.01798620996209156, 0.0024726231566347743)
```

Plot der Logistischen Funktion

In [21]:
```
spacing = np.linspace(-7, 7, 100)
plt.plot(spacing, np.array([sigmoid(t) for t in spacing]));
plt.xlabel('t')
plt.ylabel('sigmoid(t)');
```



Logistische Hypothese

Die Logistische Hypothese für 2 Eingabevariablen x1, x2 ist wie folgt definiert:

$$h(x_1, x_2) = sigmoid(x_1 w_1 + x_2 w_2 + b)$$

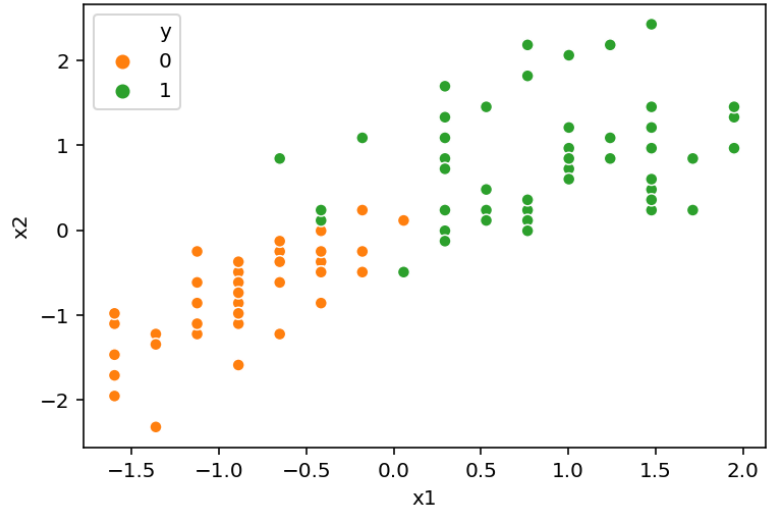
In [23]:
```
w1, w2, b = 3.3, 3.7, 0.3
h = make_logistic_hypothesis(w1, w2, b)  # Implementierung in der Übung
```

In [24]:
```
x1, x2 = -1, -1
h(x1, x2)
```

Out[24]:
```
0.0012293986212774202
```

Vergleich mit Plot

In [25]:
```
sns.scatterplot(data=df_scaled, x='x1', y='x2', hue='y', palette=cp[1:3]);
```



In [26]:
```
h(-1, -1), h(0, 1)
```

Out[26]:
```
(0.0012293986212774202, 0.9820137900379085)
```

Entscheidungsgrenze zur Klassifikation

Die Entscheidungsgrenze legt fest, ab welchem Ausgabewert der Hypothese $h(x_1, x_2)$ ein Datensatz der Klasse 0 oder 1 zugeordnet wird.

Für einfache Klassifikationsprobleme kann der Wert $threshold = 0.5$ gewählt werden, da es der Mittelwert von 0 und 1 ist.

$$classify(x_1, x_2) \begin{cases} 1, & \text{if } h(x_1, x_2) > threshold \\ 0, & \text{otherwise} \end{cases}$$

In [28]:
```
w1, w2, b = 3.3, 3.7, 0.3
threshold = 0.5
classify = make_classify(w1 ,w2, b, threshold)  # Implementierung in der Übung
```

In [29]:
```
h(-1, -1), classify(-1, -1)
```

Out[29]:
```
(0.0012293986212774202, 0)
```

In [30]:
```
h(0, 1), classify(0, 1)
```

Out[30]:
```
(0.9820137900379085, 1)
```

Entscheidungsgrenze als Funktion

Um die Entscheidungsgrenze zu erhalten setzen wir die Hypothese gleich dem Grenzwert (threshold), z.B. 0.5.

$$
\begin{aligned}
sigmoid(x_1 w_1 + x_2 w_2 + b) &= threshold \\
\Leftrightarrow x_1 w_1 + x_2 w_2 + b &= sigmoid^{-1}(threshold) \\
\Leftrightarrow x_2 &= (sigmoid^{-1}(threshold) - x_1 w_1 - b) \cdot \frac{1}{w_2} \\
\Leftrightarrow x_2 &= (ln(\frac{threshold}{1 - threshold}) - x_1 w_1 - b) \cdot \frac{1}{w_2} \\
\Rightarrow boundary(x1) &= (ln(\frac{threshold}{1 - threshold}) - x_1 w_1 - b) \cdot \frac{1}{w_2}
\end{aligned}
$$

In [31]:
```
def make_decision_boundary(w1, w2, b, threshold):
    def decision_boundary(x1):
        return (np.log(threshold / (1 - threshold)) - x1*w1 - b) * (1 / w2)

    return decision_boundary
```

Anwendung der Entscheidungsgrenze

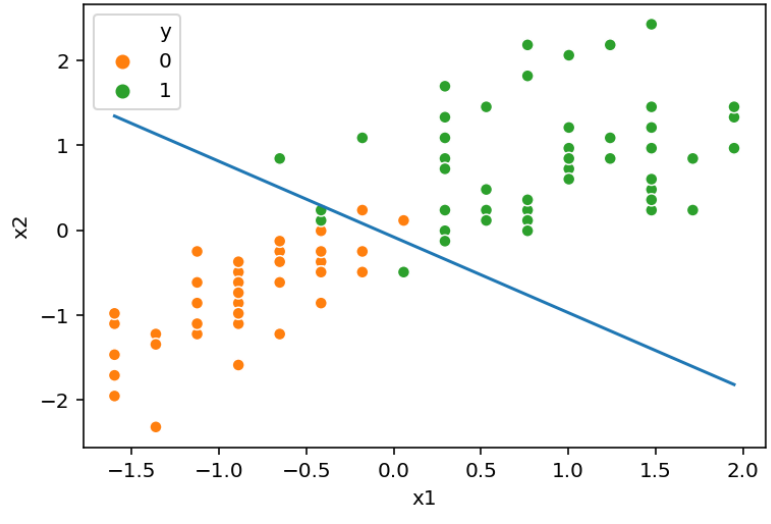
In [32]:
```
w1, w2, b = 3.3, 3.7, 0.3
threshold = 0.5

decision_boundary = make_decision_boundary(w1, w2, b, threshold)
```

In [33]:
```
decision_boundary(-1), decision_boundary(0)
```

Out[33]:
```
(0.8108108108108107, -0.08108108108108107)
```

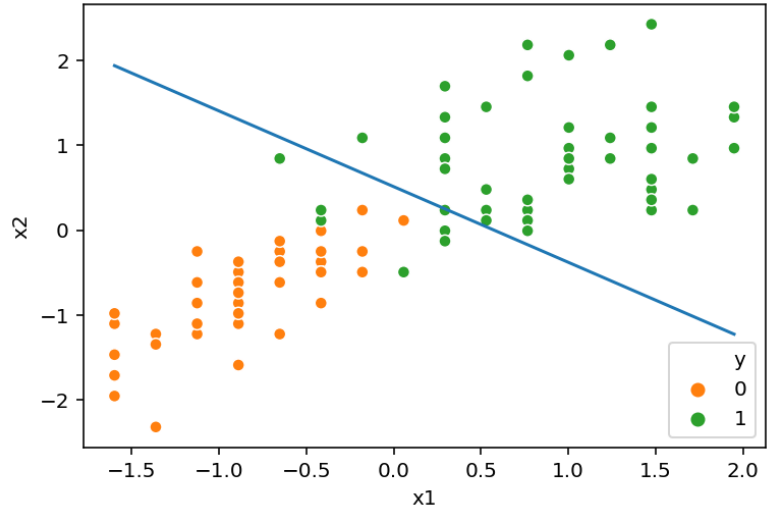
Plot der Entscheidungsgrenze

In [35]:
```
plot_boundary(df_scaled, decision_boundary)
```



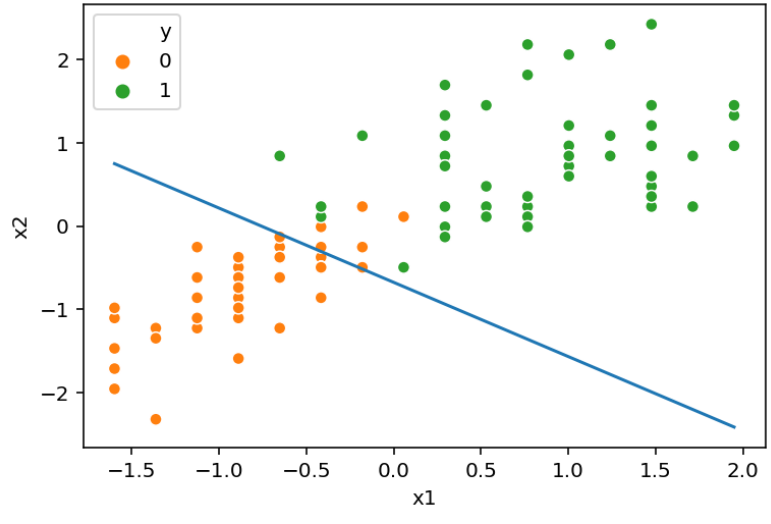
Plot der Entscheidungsgrenze mit Threshold 0.9

In [36]:
```
plot_boundary(df_scaled, make_decision_boundary(w1, w2, b, 0.9))
```



Plot der Entscheidungsgrenze mit Threshold 0.1

In [37]:
```
plot_boundary(df_scaled, make_decision_boundary(w1, w2, b, 0.1))
```



Kostenfunktion Binary-Crossentropy

Die Kostenfunktion für die Logistische Regression heißt Binary-Crossentropy und ist wie folgt defininiert.

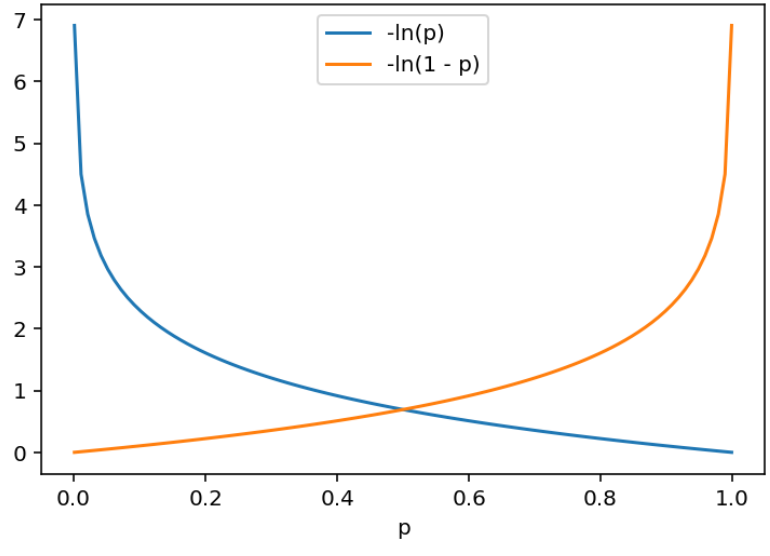
$$J(w1, w2, b) = \frac{1}{m} \sum_{i=1}^{m} y^i \cdot -ln(h(x_1^i, x_2^i)) + (1 - y^i) \cdot -ln(1 - h(x_1^i, x_2^i))$$

$$\Rightarrow J(w1, w2, b) = \frac{1}{m} \sum_{i=1}^{m} y^i \cdot -ln(p^i) + (1 - y^i) \cdot -ln(1 - p^i)$$

Zur besseren Übersicht wurde $h(x_1^i, x_2^i)$ durch p^i (prediction) ersetzt.

Plot der negativen Logarithmen

In [38]:
```
P = np.linspace(0.001, 0.999, 100)
plt.xlabel('p')
plt.plot(P, -np.log(P), label='-ln(p)')
plt.plot(P, -np.log(1 - P), label='-ln(1 - p)')
plt.legend();
```



- Für $y = 0$ berechne $-ln(1 - p)$.
- Für $y = 1$ berechne $-ln(p)$.

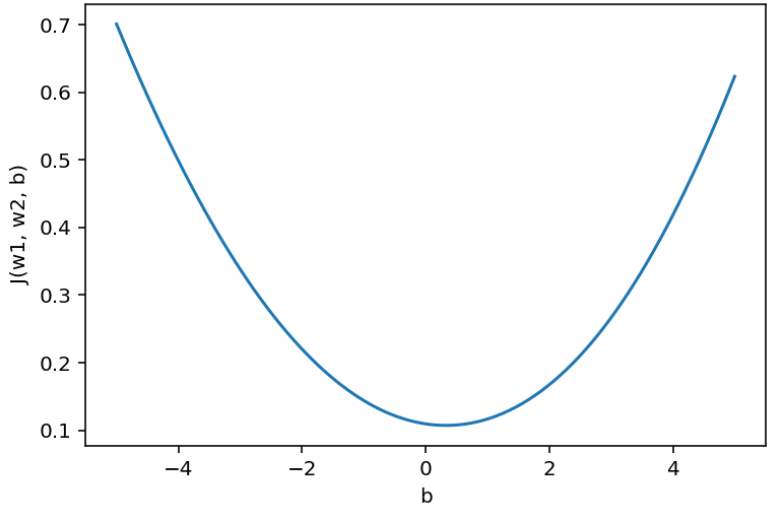
Binary Crossentropy in Python

In [40]:
```
J = make_binary_crossentropy_cost(X_scaled, Y)  # Implementierung in der Übung
w1, w2, b = 3.3, 3.7, 0.3
J(w1, w2, b)
```

Out[40]:
```
0.10736084146625315
```

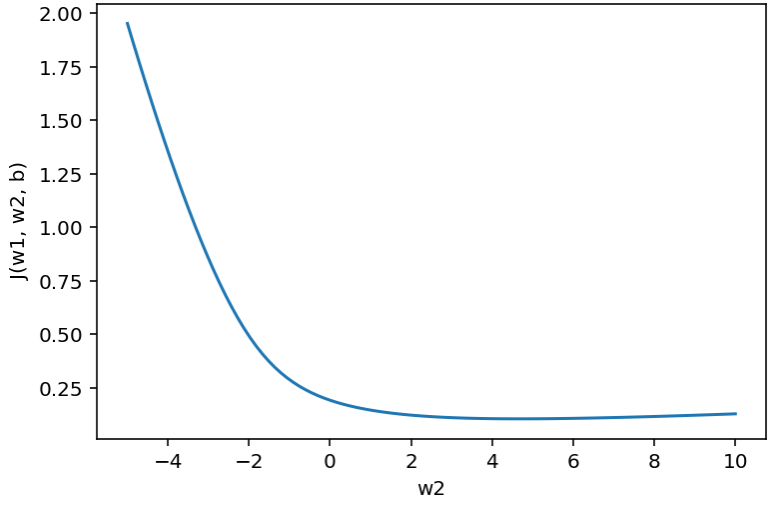
Plot der Kostenfunktion abhängig von b

In [41]:
```
spacing = np.linspace(-5, 5, 100)
costs = [J(3.3, 3.7, b) for b in spacing]

plt.plot(spacing, costs)
plt.xlabel('b')
plt.ylabel('J(w1, w2, b)');
```



Plot der Kostenfunktion abhängig von w2

In [42]:
```
spacing = np.linspace(-5, 10, 100)
costs = [J(3.3, w2, 0.3) for w2 in spacing]

plt.plot(spacing, costs)
plt.xlabel('w2')
plt.ylabel('J(w1, w2, b)');
```



Partielle Ableitungen

Die Ergebnisse der partiellen Ableitungen nach b, w_1 und w_2 lauten

$$\frac{\partial}{\partial b} J(w_1, w_2, b) = \frac{1}{m} \sum_{i=1}^{m} (h(x_1^i, x_2^i) - y^i)$$

$$\frac{\partial}{\partial w_1} J(w_1, w_2, b) = \frac{1}{m} \sum_{i=1}^{m} (h(x_1^i, x_2^i) - y^i) \cdot x_1^i$$

$$\frac{\partial}{\partial w_2} J(w_1, w_2, b) = \frac{1}{m} \sum_{i=1}^{m} (h(x_1^i, x_2^i) - y^i) \cdot x_2^i$$

mit $x_i \in X, y_i \in Y$ und $m = |Y| = |X|$.

Berechnung der Gradienten

In [44]:
```
gradient = make_gradient(X_scaled, Y)  # Implementierung in der Übung

w1, w2, b = 3.3, 3.7, 0.3
pd_w1, pd_w2, pd_b = gradient(w1, w2, b)
pd_w1, pd_w2, pd_b
```

Out[44]:
```
(-0.005782754927171705, -0.003701327840293857, -0.0012556861902088933)
```

Gradientenabstiegsverfahren

Pseudocode:

Initialisiere w1, w2 und b zufällig.

Für eine Anzahl an Epochen wiederhole:

$$
\begin{aligned}
pd\_w1 &:= \frac{\partial}{\partial w_1} J(w_1, w_2, b) \\
pd\_w2 &:= \frac{\partial}{\partial w_2} J(w_1, w_2, b) \\
pd\_b &:= \frac{\partial}{\partial b} J(w, b)
\end{aligned}
$$

$$
\begin{aligned}
w1 &:= w1 - \alpha * pd\_w1 \\
w2 &:= w2 - \alpha * pd\_w2 \\
b &:= b - \alpha * pd\_b
\end{aligned}
$$

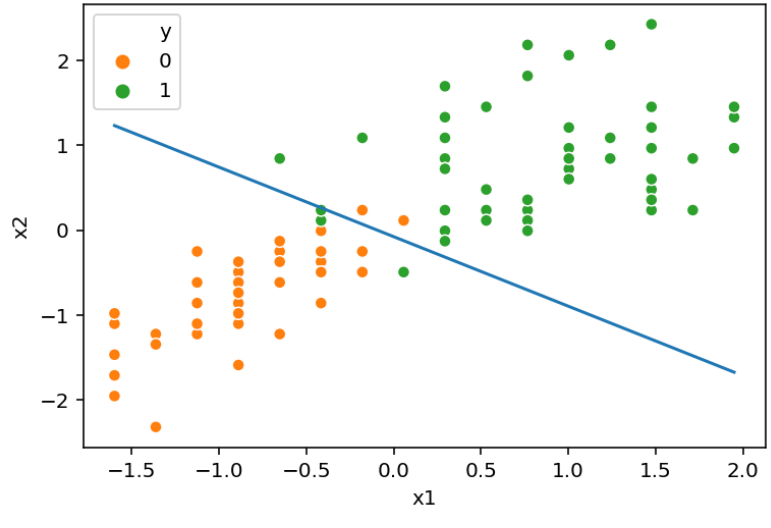
Die Lernrate α bestimmt die Schrittgröße der Updates und ist ein Wert größer Null, üblicherweise im Bereich $]0, 1]$.

Anwendung von SGD

In [46]:
```
alpha = 0.1
epochs = 1500
w1, w2, b = np.random.randn(3)
w1, w2, b, cost_per_epoch = sgd(X_scaled, Y, w1, w2, b, alpha, epochs)  # Imple
mentierung in der Übung
w1, w2, b
```

Out[46]:
```
(3.1198404790362497, 3.808190507383827, 0.2939813322700435)
```

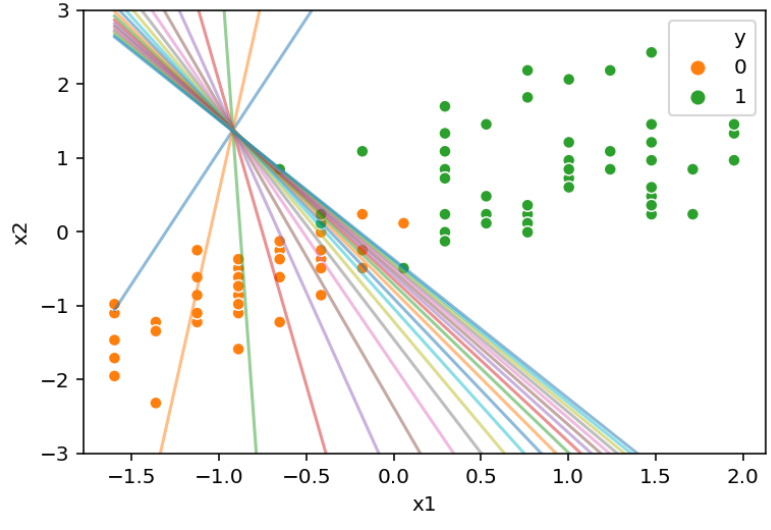
In [47]:
```
plot_boundary(df_scaled, make_decision_boundary(w1, w2, b, 0.5))
```



Entscheidungsgrenzen im Trainingsverlauf

In [49]:

```
np.random.seed(42)
w1, w2, b = np.random.randn(3)
plot_training_boundaries(w1, w2, b, alpha=0.25, epochs=20)
```



Plot der Kosten pro Epoche

In [51]:
```
plot_over_time(cost_per_epoch)
```