Machine Learning: Lineare Regression

Präsentation: Vorname, Nachname

Lizenz:

deep.TEACHING

www.deep-teaching.org

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**?

Python

Python ist die wichtigste High-Level-Sprache im Machine Learning-Umfeld.

- Explorativer Ansatz durch dynamisches Ausführen von Code (Jupyter)
- Machine Learning ist Rechenintensiv, Python nutzt hochoptimierte Bibliotheken im Hintergrund
  - Erspart dem Anwender das Programmieren in C, C++, Fortran, Cuda, OpenCL, ...
- Konkurrierende Sprachen: R, Lua, Matlab (Octave), Julia, JavaScript, ...

Auszug des Python Ökosystems

Datenstrukturen	**Numpy**, Scipy, **Pandas**
Plotting	**Matplotlib**, **Seaborn**, Graphviz
Klassisches Machine Learning	**Scikit-Learn**
Bildverarbeitung	Pillow, Scikit-Image
Computational Graphs (low-level)	Tensorflow, Theano, PyTorch
Künstliche Neuronale Netze (high-level)	Keras (nutzt low-level), PyTorch

Fishers Iris-Daten

In [4]:
```
df.head()
```

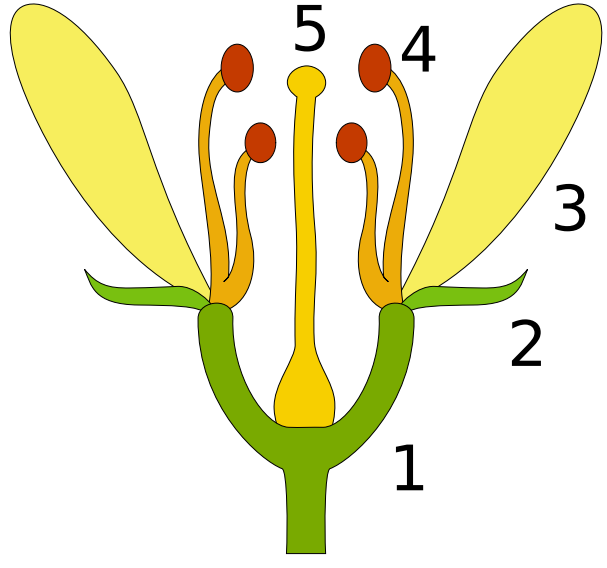
Out[4]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In [5]:
```
df['species'].value_counts() # Schwertlilien
```

Out[5]:
```
Iris-versicolor    50
Iris-setosa        50
Iris-virginica     50
Name: species, dtype: int64
```

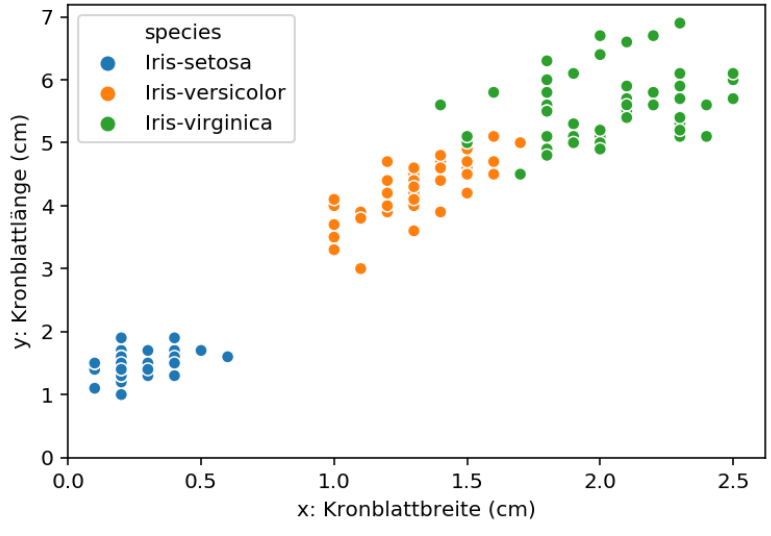
Blumen?



1. Kelchblätter (Sepalen)
2. Kronblätter (Petalen)

Plot der Daten

In [6]:
```
sns.scatterplot(data=df, x='petal_width', y='petal_length', hue='species')
plt.xlabel('x: Kronblattbreite (cm)'); plt.ylabel('y: Kronblattlänge (cm)')
plt.xlim(0, None); plt.ylim(0, None);
```



Angenommen eine Blüte weist eine Breite von 1,5 cm auf, wie lang ist sie vermutlich?

X- und Y-Arrays

In [7]:
```
X = df['petal_width'].values
Y = df['petal_length'].values
```

In [8]:
```
X[:5]
```

Out[8]:
```
array([0.2, 0.2, 0.2, 0.2, 0.2])
```

In [9]:
```
Y[:5]
```

Out[9]:
```
array([1.4, 1.4, 1.3, 1.5, 1.4])
```

In [10]:
```
X.shape, Y.shape
```

Out[10]:
```
((150,), (150,))
```

Lineare Hypothese

Geradengleichung ist

$$h(x) = x \cdot w + b$$

mit

- w als Steigung (weight),
- b als Y-Achsenabschnitt (bias),
- und x als Eingabe (feature)

Lineare Hypothese in Python

$$h(x) = x \cdot w + b$$

In [11]:
```
def make_linear_hypothesis(w, b):
    # this is a closure
    def linear_hypothesis(x):
        return x * w + b

    return linear_hypothesis

h = make_linear_hypothesis(2.5, 1)
h(1.5)  # prediction
```

Out[11]:
```
4.75
```

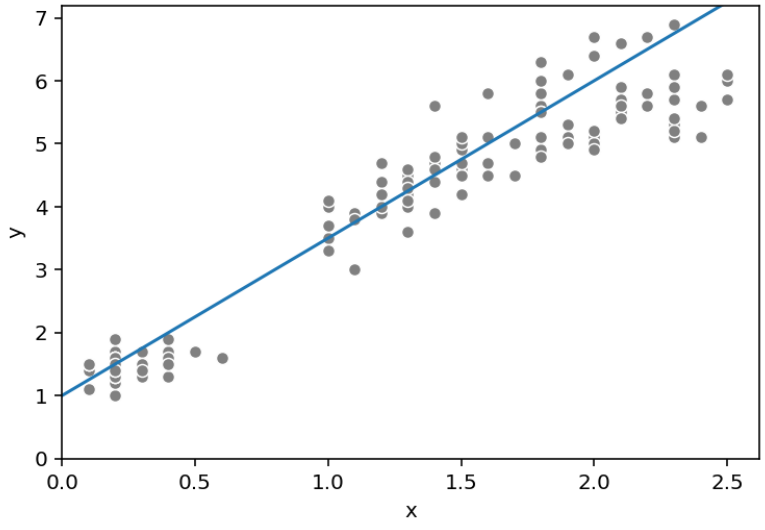
Plot der Hypothese

In [12]:
```
df_reduced = pd.DataFrame(X, columns=['x'])
df_reduced['y'] = Y
df_reduced.head()
```

Out[12]:

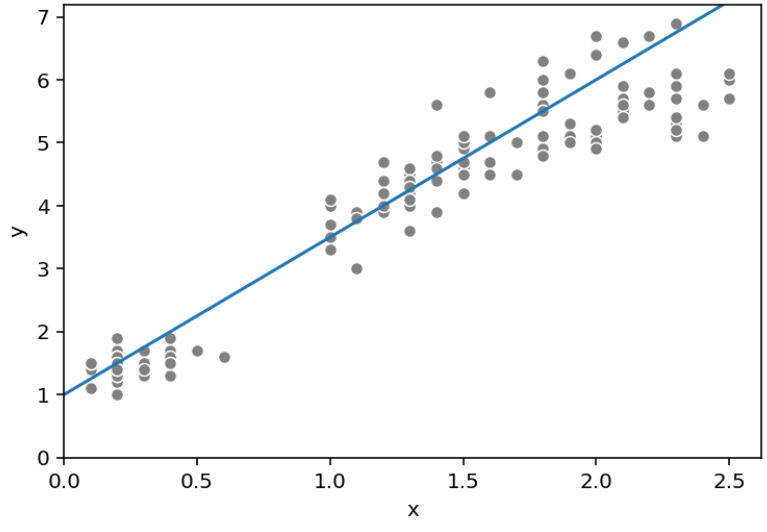
	x	y
0	0.2	1.4
1	0.2	1.4
2	0.2	1.3
3	0.2	1.5
4	0.2	1.4

In [14]:
```
plot(df_reduced, make_linear_hypothesis(2.5, 1))
```



Vorhersage

In [15]:
```
plot(df_reduced, make_linear_hypothesis(2.5, 1))
```



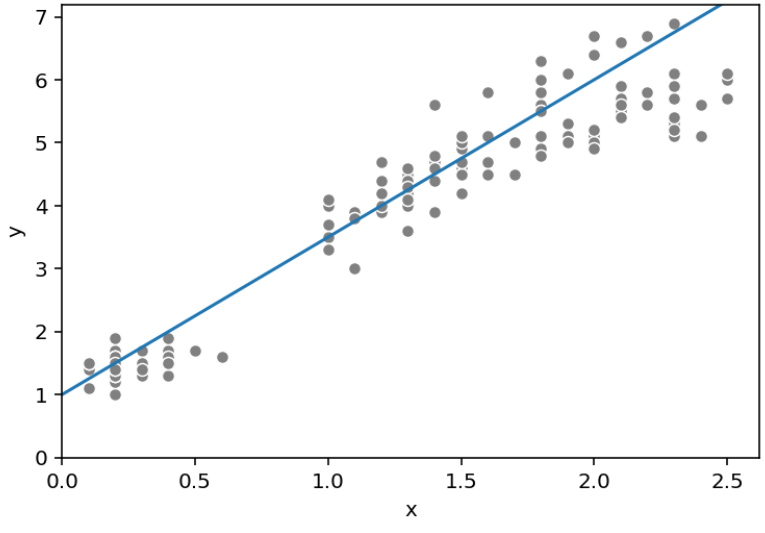
Eine Vorhersage (prediction) der Kronblattlänge gegeben der Kronblattbreite.

In [16]:
```
h = make_linear_hypothesis(2.5, 1)
h(0.25), h(1.5), h(2)
```

Out[16]:
```
(1.625, 4.75, 6.0)
```

Vorhersage

In [17]:
```
h = make_linear_hypothesis(2.5, 1)
plot(df_reduced, h)
```



Wir wissen nicht ob sich der Trend außerhalb der ursprünglichen Daten fortsetzt.

In [18]:
```
h(-1), h(0.75), h(3)
```

Out[18]:
```
(-1.5, 2.875, 8.5)
```

Lineare Regression

Ziel: Finde mathematisches Modell für die möglichst genaue Vorhersage (prediction) kontinuierlicher Werte.

Benötigte Komponenten:

1. Hypothese (hypothesis) / Modell zur Vorhersage
2. Fehlerfunktion (cost / loss function) zum Lernen / Trainieren und zur Evaluation
3. Optimierungsalgorithmus (optimizer) zum Lernen / Trainieren

Fehlerfunktion: Mean Squared Error

Die für eine Lineare Regression übliche Fehlerfunktion J ist Mean Squared Error. Die mathematische Definition lautet

$$J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} (h_{w,b}(x_i) - y_i)^2$$

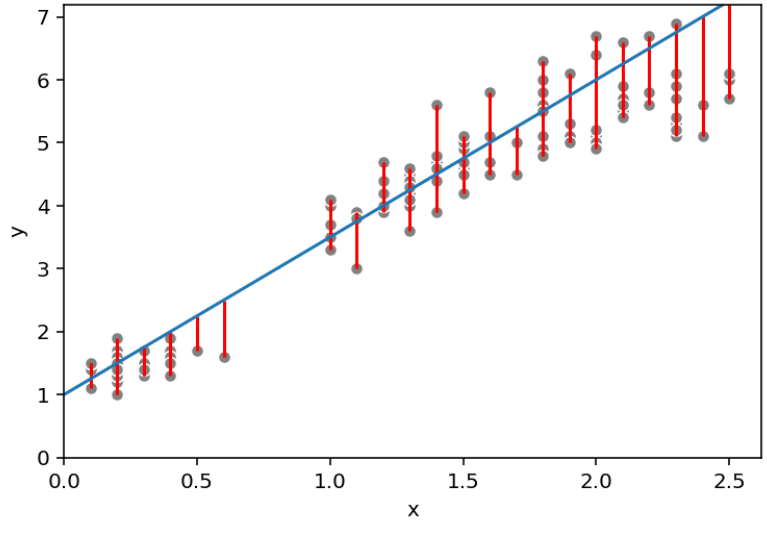
mit $x_i \in X$, $y_i \in Y$ und $m = |Y| = |X|$.

Für jedes x_i und y_i wird der Abstand zwischen $h(x_i)$ und y_i berechnet. Anschließend werden diese Abstände (Fehler) quadriert und der Mittelwert gebildet.

Durch $2m$ zu teilen, statt durch m, ist ein mathematischer Trick, der später noch relevant wird.

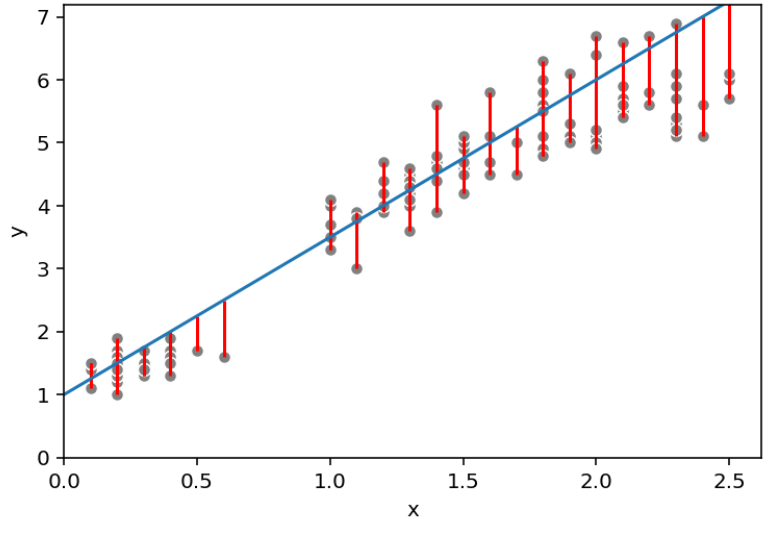
Visualisierung der Fehler

In [19]:
```
plot(df_reduced, make_linear_hypothesis(2.5, 1), show_errors=True)
```



Fehlerberechnung

In [20]:
```
plot(df_reduced, make_linear_hypothesis(2.5, 1), show_errors=True)
```

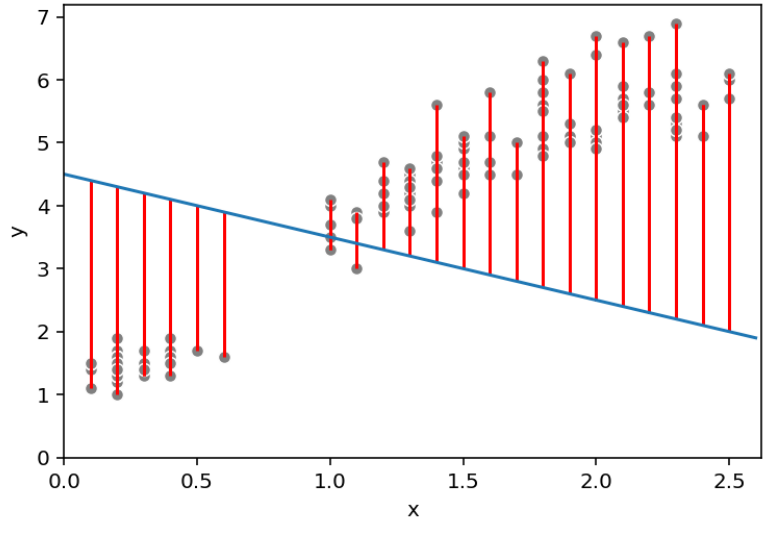


In [22]:
```
J = make_mse_cost(X, Y)  # Implemetierung in der Übung
J(2.5, 1)                # Mean Squared Error
```

Out[22]:
```
0.16308333333333333
```

Fehlerberechnung für schlechte Parameter

In [23]:
```
plot(df_reduced, make_linear_hypothesis(-1, 4.5), show_errors=True)
```



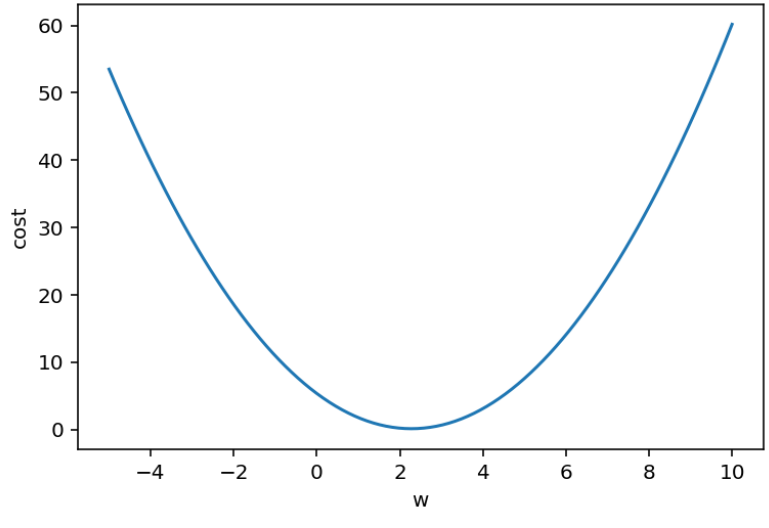
In [24]:
```
J(-1, 4.5)
```

Out[24]:
```
3.2278
```

Visualisierung der Kostenfunktion

Wert von b ist fix, w ist variabel.

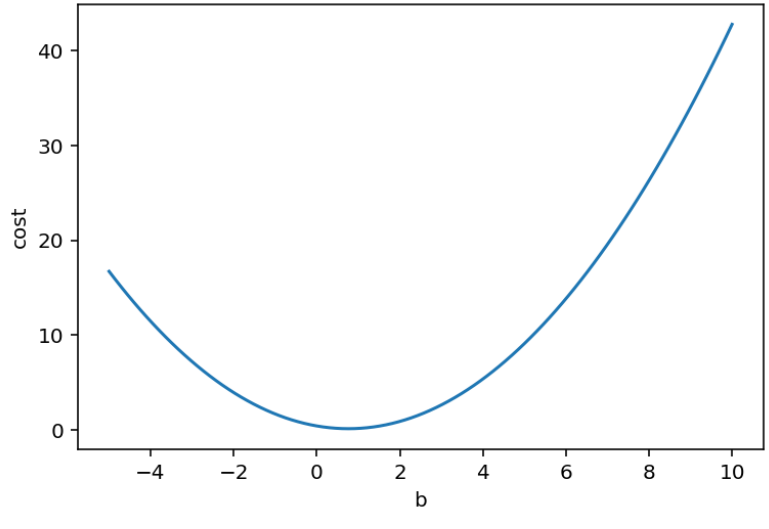
In [25]:
```
spacing = np.linspace(-5, 10, 100)
costs = [J(w, 1) for w in spacing]

plt.plot(spacing, costs)
plt.xlabel('w')
plt.ylabel('cost');
```



Visualisierung der Kostenfunktion

Wert von w ist fix, b ist variabel.

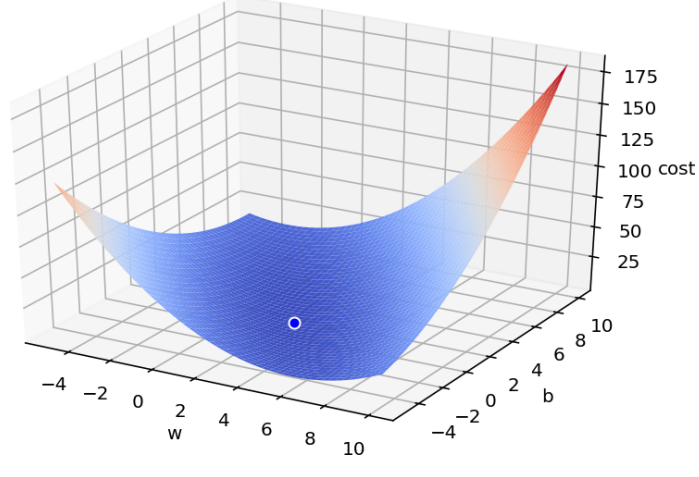
In [26]:
```
spacing = np.linspace(-5, 10, 100)
J(X, Y)
costs = [J(2.5, b) for b in spacing]

plt.plot(spacing, costs)
plt.xlabel('b')
plt.ylabel('cost');
```



3D Visualisierung der Kostenfunktion

- Bei nicht optimalen Werten der Variablen w oder b, ergibt sich für die jeweils andere Variable eine verzerrte Kostenfunktion (z.B. an der Stelle $b = -7.5$).
- Je nach Ausgangspunkt kann dies den Optimierungsalgorithmus verlangsamen.

In [28]:
```
plot_cost_3d(w_range=(-5, 10), b_range=(-5, 10), w_opt=2.5, b_opt=1)
```



Skalierung der Eingabevariable

Der `StandardScaler` aus Scikit-Learn skaliert X, indem

- der Mittelwert abgezogen und
- durch die Standardabweichung geteilt wird,

sodass

- die Daten einen Mittelwert von 0 und
- eine Standardabweichung von 1 haben.

Skalierung der Eingabevariable

In [29]:
```
scaler = StandardScaler()  # import aus Scikit-Learn
X_scaled = scaler.fit_transform(X.reshape(-1, 1))
X_scaled = X_scaled.flatten()
```

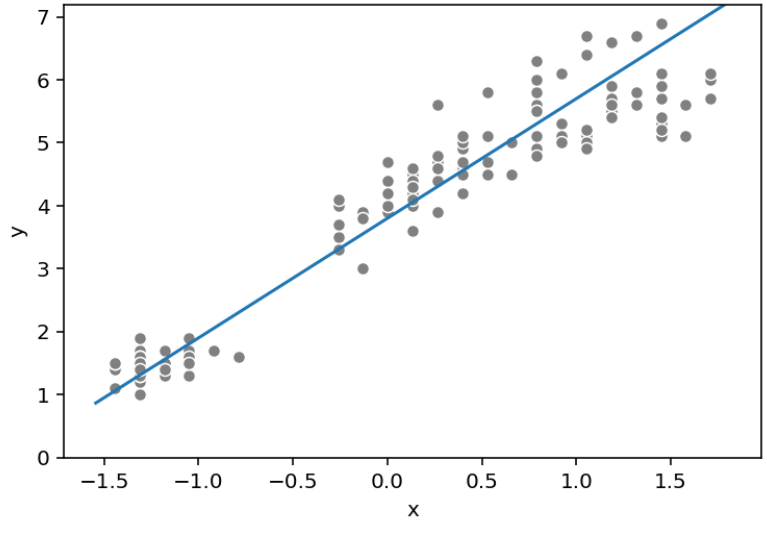
In [31]:
```
comparison()
```

Out[31]:

	min	max	mean	std
X	0.10	2.50	1.2	0.76
X_scaled	-1.44	1.71	-0.0	1.00

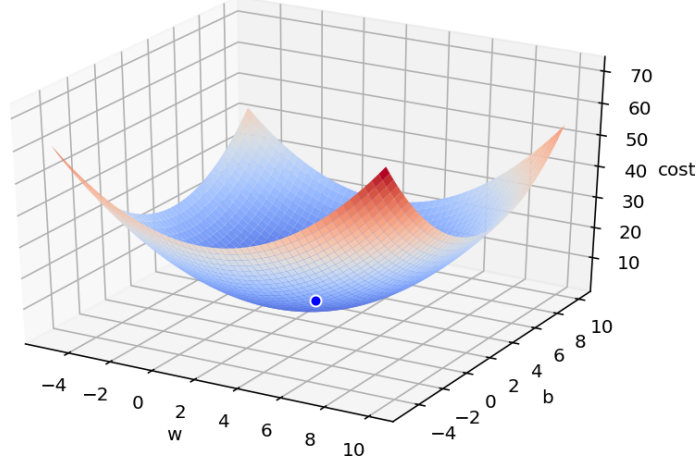
Plot der skalierten Daten

In [32]:
```
df_scaled = pd.DataFrame(X_scaled, columns=['x'])
df_scaled['y'] = Y

plot(df_scaled, make_linear_hypothesis(1.9, 3.8))
```



3D Visualisierung nach Skalierung

In [33]:
```
J = make_mse_cost(X_scaled, Y)
plot_cost_3d(w_range=(-5, 10), b_range=(-5, 10), w_opt=1.9, b_opt=3.8)
```



Optimierung

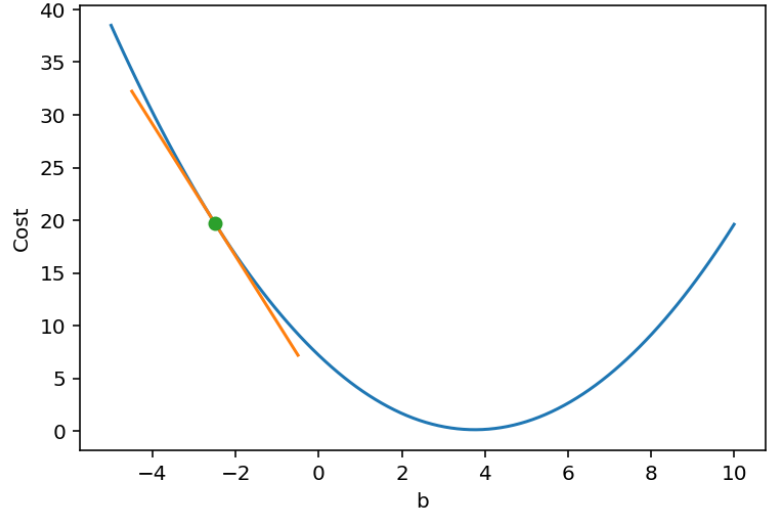
Ziel: Finde automatisiert Werte für w und b, die einen minimalen Fehlerwert liefern.

- Startpunkt ist eine zufällige Initialisierung von w und b.
- Iterative Anwendung des Gradientenabstiegsverfahrens (Stochastic Gradient Descent) zur schrittweisen Annäherung an das Minimum.
  - Berechnung der partiellen Ableitungen (Steigungen) von J an einem Punkt (w_i, b_i) in die Richtungen w und b.
  - Aktualisierung von w und b anhand des Ableitungswertes.

Visualisierung der Ableitung nach b

In [34]:
```
w = 1.9
b = -2.5
```

In [36]:
```
plot_b_derivative()
```



Partielle Ableitungen

Die Ergebnisse der partiellen Ableitungen nach b und w lauten

$$\frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} (h_{w,b}(x_i) - y_i) \cdot x_i$$

$$\frac{\partial}{\partial b} J(w,b) = \frac{1}{m} \sum_{i=1}^{m} (h_{w,b}(x_i) - y_i)$$

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

Erklärung zur Berechnung der Ableitung: mccormickml.com/2014/03/04/gradient-descent-derivation/ (http://mccormickml.com/2014/03/04/gradient-descent-derivation/)

Partielle Ableitung nach b

In [38]:
```
w = 1.9
gradient = make_gradient(X_scaled, Y)  # Implementierung in der Übung

for b in range(-5, 10):
    pd_w, pd_b = gradient(w, b)  # calculates partial derivatives
    print(pd_b)
```

```
-8.758666666666667
-7.7586666666666675
-6.758666666666667
-5.758666666666667
-4.758666666666667
-3.758666666666667
-2.7586666666666666
-1.758666666666667
-0.7586666666666669
0.24133333333333315
1.241333333333333
2.2413333333333334
3.2413333333333334
4.241333333333333
5.241333333333333
```

Partielle Ableitung nach w

In [39]:
```
b = 3.8
gradient = make_gradient(X_scaled, Y)

for w in range(-5, 10):
    pd_w, pd_b = gradient(w, b)
    print(pd_w)
```

```
-6.693036451694907
-5.693036451694907
-4.693036451694907
-3.6930364516949057
-2.6930364516949057
-1.693036451694906
-0.6930364516949057
0.3069635483050944
1.3069635483050943
2.3069635483050948
3.3069635483050948
4.306963548305095
5.306963548305095
6.306963548305095
7.306963548305095
```

Gradientenabstiegsverfahren

Stochastic Gradient Descent (SGD)

Pseudocode:

Initialisiere w und b zufällig.

Für eine Anzahl an Epochen wiederhole:

$$pd\_w := \frac{\partial}{\partial w} J(w, b)$$
$$pd\_b := \frac{\partial}{\partial b} J(w, b)$$

$$w := w - \alpha * pd\_w$$
$$b := b - \alpha * pd\_b$$

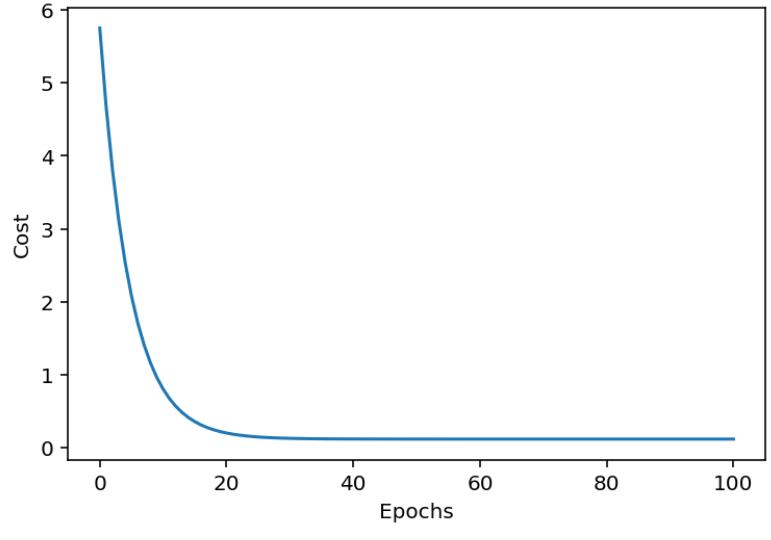
Die Lernrate α bestimmt die Schrittgröße der Updates und ist ein Wert größer Null, üblicherweise im Bereich $]0, 1]$.

Anwendung von SGD

In [42]:
```
alpha = 0.1
epochs = 100
w, b = np.random.randn(2)
w, b, cost_per_epoch = sgd(X_scaled, Y, w, b, alpha, epochs)  # Implementierung
 in der Übung
w, b
```

Out[42]:
```
(1.6930383379764007, 3.758577459957277)
```

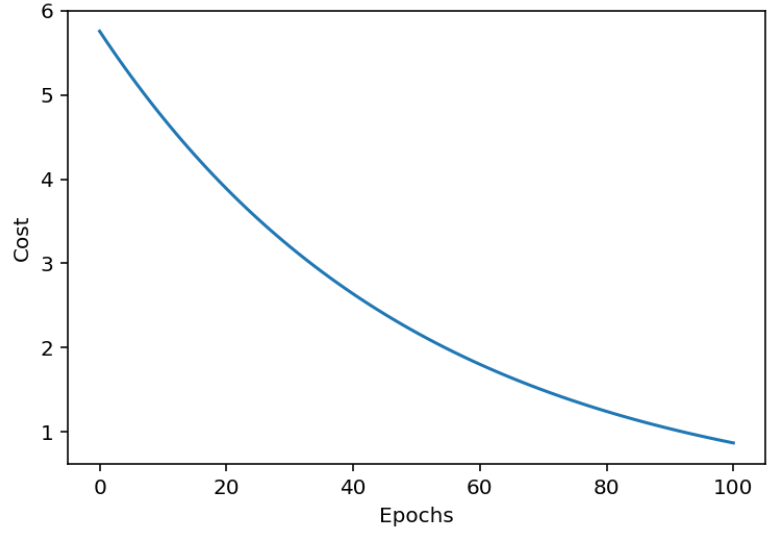
In [44]:
```
plot_over_time(cost_per_epoch)  # Implementierung in der Übung
```



Zu kleine Lernrate

Das Training endet bevor der Kostenwert konvergiert ist.

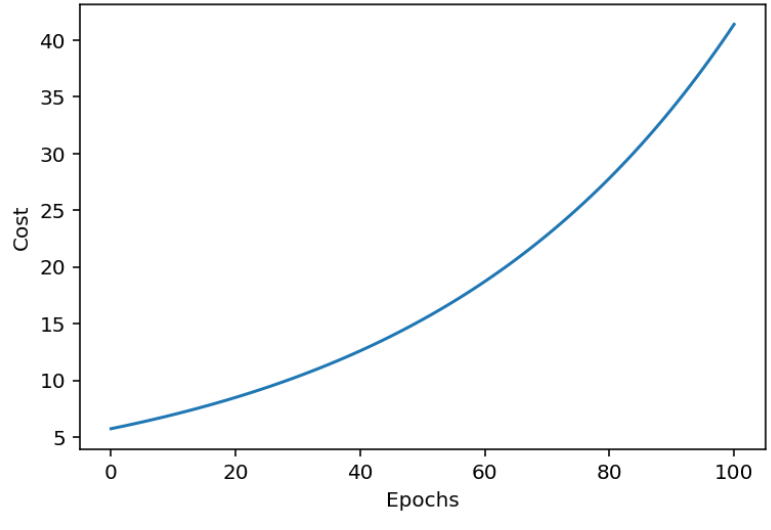
In [46]:
```
alpha = 0.01
epochs = 100
w, b = np.random.randn(2)
w, b, cost_per_epoch = sgd(X_scaled, Y, w, b, alpha, epochs)

plot_over_time(cost_per_epoch)
```



Zu große Lernrate

Der Kostenwert wird schlechter, weil das Update von w und b über das Ziel hinaus schießt.

In [48]:
```
alpha = 2.01
epochs = 100
w, b = np.random.randn(2)
w, b, cost_per_epoch = sgd(X_scaled, Y, w, b, alpha, epochs)

plot_over_time(cost_per_epoch)
```



Machine Learning Algorithmen

Das Gradientenabstiegsverfahren ist einer von vielen Machine Learning-Algorithmen zur iterativen **Optimierung** eines mathematischen **Modells** anhand von **Trainingsdaten**.

- **Lineare Regression** $\rightarrow$ Lösung von Regressionsproblemen
- **Logistische Regression** $\rightarrow$ Lösung von Klassifikationsproblemen (der Name ist irreführend)
  - wie Lineare Regression, aber mit anderer Kostenfunktion
- **Künstliche Neuronale Netze** $\rightarrow$ Lösung komplexer Regressions- und Klassifikationsprobleme
  - wie Lineare / Logistische Regression, aber größere / vielschichtige Modelle

Supervised und Unsupervised Learning

Die genannten Regressions- und Klassifikationsalgorithmen sind **supervised** Learning Algorithmen, weil die gesuchten **Zielwerte** Y für die Trainingsdaten **bekannt** sind.

Beispiel für Supervised Learning:

- Hausklassifikation: Klassen **renoviert** und **nicht renoviert** sind für Trainingsdaten bekannt.

Beispiel für Unsupervised Learning:

- Clustern von Bildern in 5 möglichst unterschiedliche Farbkategorien, wobei vorher nicht bekannt ist, welche Farben die Clusternzentren bilden werden.

Wir werden in dieser Vorlesungsreihe keine unsupervised Learning Algorithmen behandeln.