Deel I. Machine learning

Hoofdstuk 2. Generieke neuronen

Inleiding

Dit is het Jupyter Notebook behorende bij hoofdstuk 2 van het vak *Advanced Datamining* (BFVH4DMN2). Op BlackBoard tref je eveneens een module `data.py` aan die diverse functies bevat die helpen bij het genereren en het visualiseren van de gebruikte datasets. Kopieer het bestand `model.py` van het vorige hoofdstuk en sla deze bestanden gezamenlijk op in één werkmap. Open je `model` module in een code-editor naar keuze om hiermee verder te werken.

Laten we weer beginnen om deze functies te importeren, samen met wat initialisatie-code en enkele onderdelen van de modules `sklearn` en `pandas`. Plaats de cursor in de cel hieronder en druk op Ctrl+Enter (of Shift+Enter om meteen naar de volgende cel te gaan).

In [1]:
```
%matplotlib inline
%reload_ext autoreload
%autoreload 2

from sys import version
print(f'Using python version {version.split(" ")[0]}')

from pandas import DataFrame, __version__
print(f'Using pandas version {__version__}')

from sklearn import linear_model, svm, __version__
print(f'Using sklearn version {__version__}')

import vlearning
from vlearning import data, __version__
from vlearning import activation_functions, loss_functions
print(f'Using vlearning version {__version__}')
```

```
Using python version 3.11.8
Using pandas version 2.2.1
Using sklearn version 1.4.1.post1
Using vlearning version 0.2.0
```

Activatie- en loss-functies

Lineaire regressie maakt gebruik van de identiteitsfunctie als activatiefunctie. Van het perceptron kennen we reeds de signum-functie. Deze functies hebben één parameter en retourneren één getalwaarde. Het zijn daarmee relatief elementaire functies.

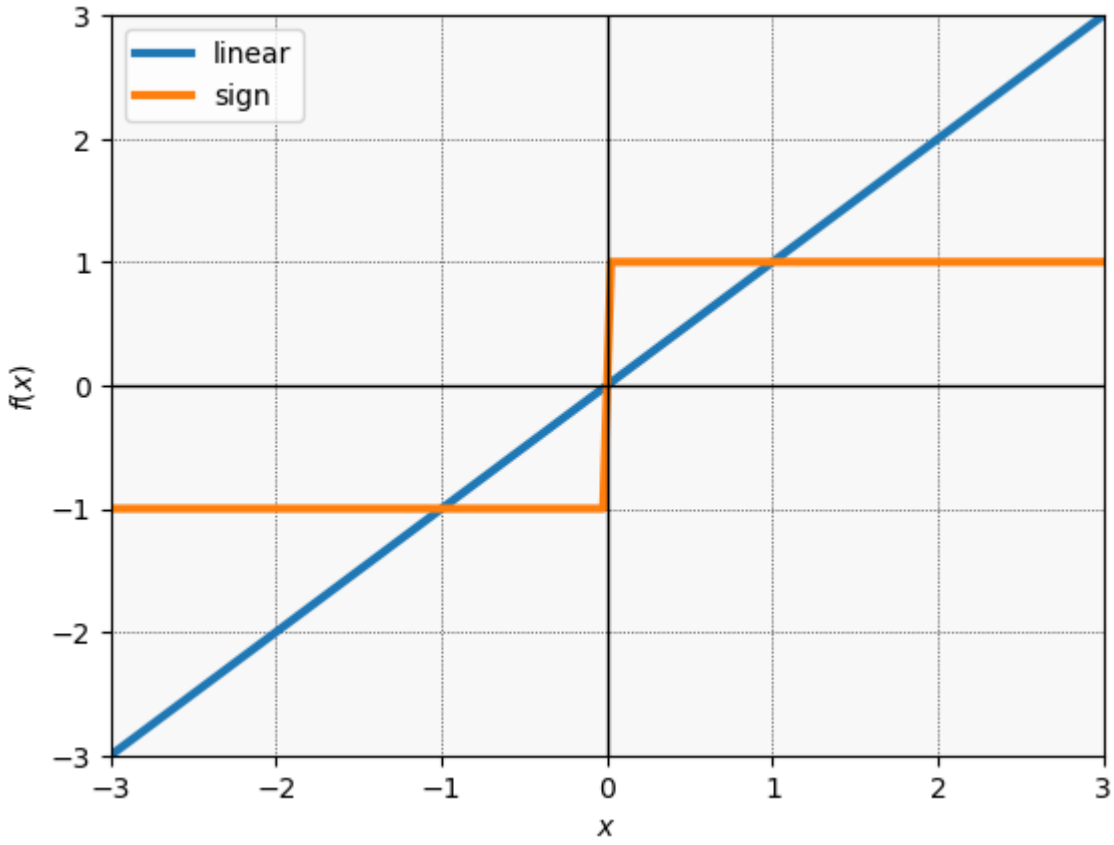
De identiteitsfunctie kan bijvoorbeeld eenvoudigweg worden geïmplementeerd als:

```
def linear(a):
    return a
```

Voeg naast deze functie ook een `sign()` functie toe aan je `model` module met de signum functie. Voel je vrij om ook enkele andere activatiefuncties te implementeren (zie bv. de tabel op Wikipedia voor een uitgebreid overzicht).

Met de code hieronder kun je je activatie-functies weergeven met behulp van de `data.graph()` functie. Controleer dat alle door jou geïmplementeerde functies de juiste verwachte vorm hebben.

In [2]:
```
data.graph([activation_functions.linear, activation_functions.sign])
```

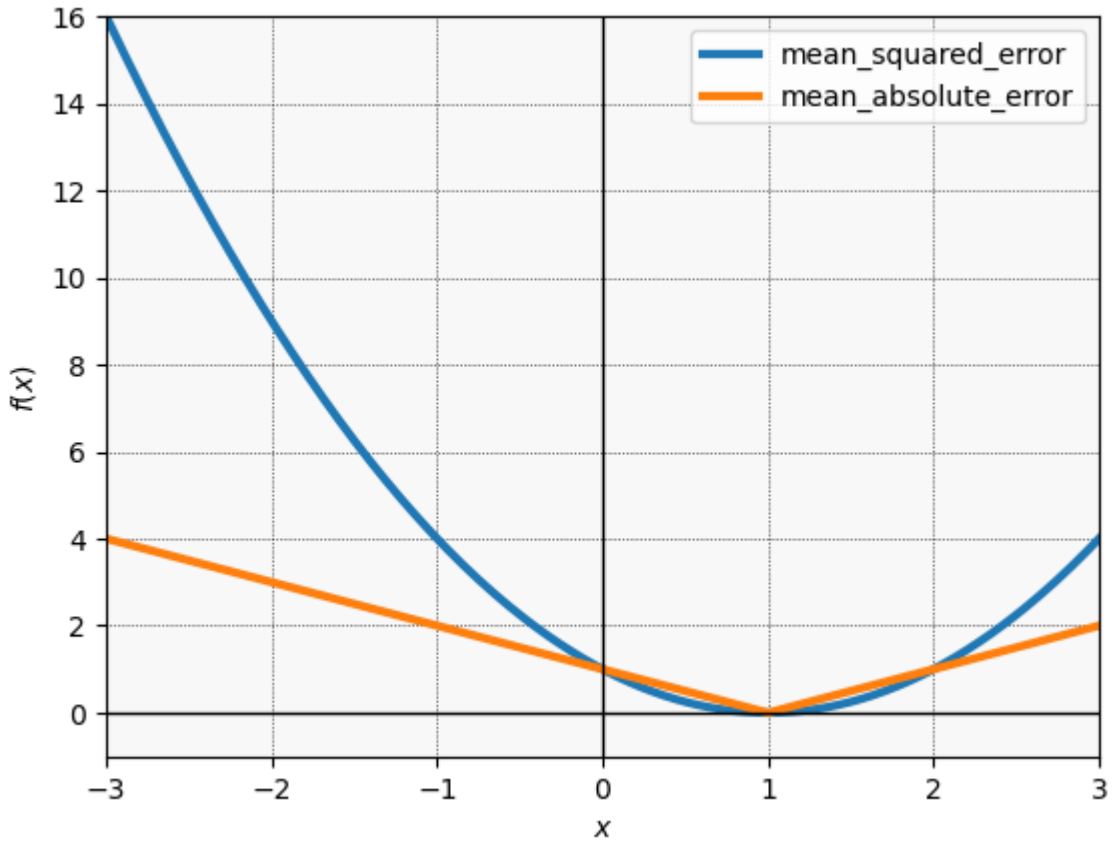


Hetzelfde kunnen we doen met de loss-functies. We zijn één voorname loss-functie tegengekomen, de kwadratische loss-functie $\mathcal{L}(\hat{y}; y) = (\hat{y} - y)^2$. Implementeer deze in de vorm van een functie `mean_squared_error(yhat, y)`. Voeg ook een variant

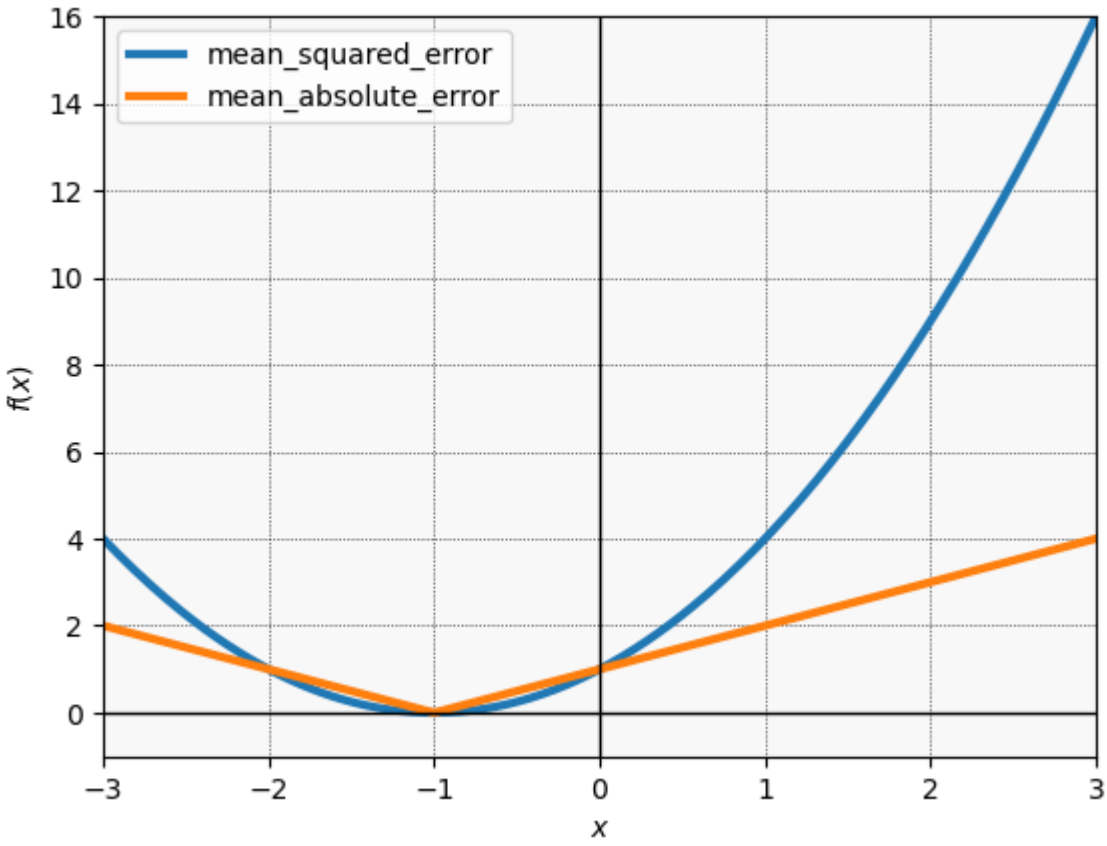
`mean_absolute_error(yhat, y)` toe die de absolute waarde van de afwijking neemt volgens $\mathcal{L}(\hat{y}; y) = |\hat{y} - y|$.

Geef met de code hieronder wederom de functie weer. Merk op dat de loss-functie twee parameters heeft: $\hat{y}$ en y. Met `data.graph()` kun je de afhankelijkheid van de eerste parameter tonen; de tweede parameter kun je een vaste waarde meegeven. Het resultaat is dus een loss-functie die aangeeft hoe groot de loss van een instance met voorspelling $\hat{y}$ is, *gegeven een bepaalde gewenste uitkomst* y. Ga na dat uit de grafiek blijkt dat de kwadratische loss-functie gelijk is aan nul als $\hat{y} = y$. Geldt dit ook voor de absolute loss-functie?

In [3]:
```
y = +1.0
data.graph([loss_functions.mean_squared_error, loss_functions.mean_absolute_
```



In [4]:
```
y = -1.0
data.graph([loss_functions.mean_squared_error, loss_functions.mean_absolute_
```



Tenslotte is het van belang dat we kunnen werken met de afgeleiden van diverse functies. Afgeleiden zijn analytisch te bepalen door te differentiëren, maar voor het gemak zullen we hier gebruik maken van numerieke afgeleiden die je bijvoorbeeld kunt bepalen met de formule:

$$\frac{\partial y}{\partial x} \approx \frac{f\left(x+\Delta x\right) - f\left(x-\Delta x\right)}{2\Delta x}$$

De waarde van Δx moet voldoende klein genomen worden om de benadering kloppend te laten zijn ter plekke van de gevraagde x. Aan de andere kant moet Δx ook voldoende groot genomen worden opdat afrondfouten niet de overhand krijgen omdat een computer getallen nou eenmaal met een beperkt aantal cijfers achter de komma representeert.

Implementeer de bovenstaande formule in de vorm van een functie `derivative()`. Deze dient de afgeleide te bepalen van een zekere functie `function()` die meegegeven wordt als argument. Een optioneel argument `delta` geeft de stapgrootte Δx aan; kies hiervoor een geschikte (kleine, maar niet te kleine) default waarde. Als return-value dient `derivative()` een functie te geven die de afgeleide van de gespecificeerde functie bevat. Dit wordt bereikt door een inner functie `wrapper_derivative()` te nesten in de body van de `derivative()` functie zelf en deze te retourneren. Dit komt er ongeveer als volgt uit te zien:

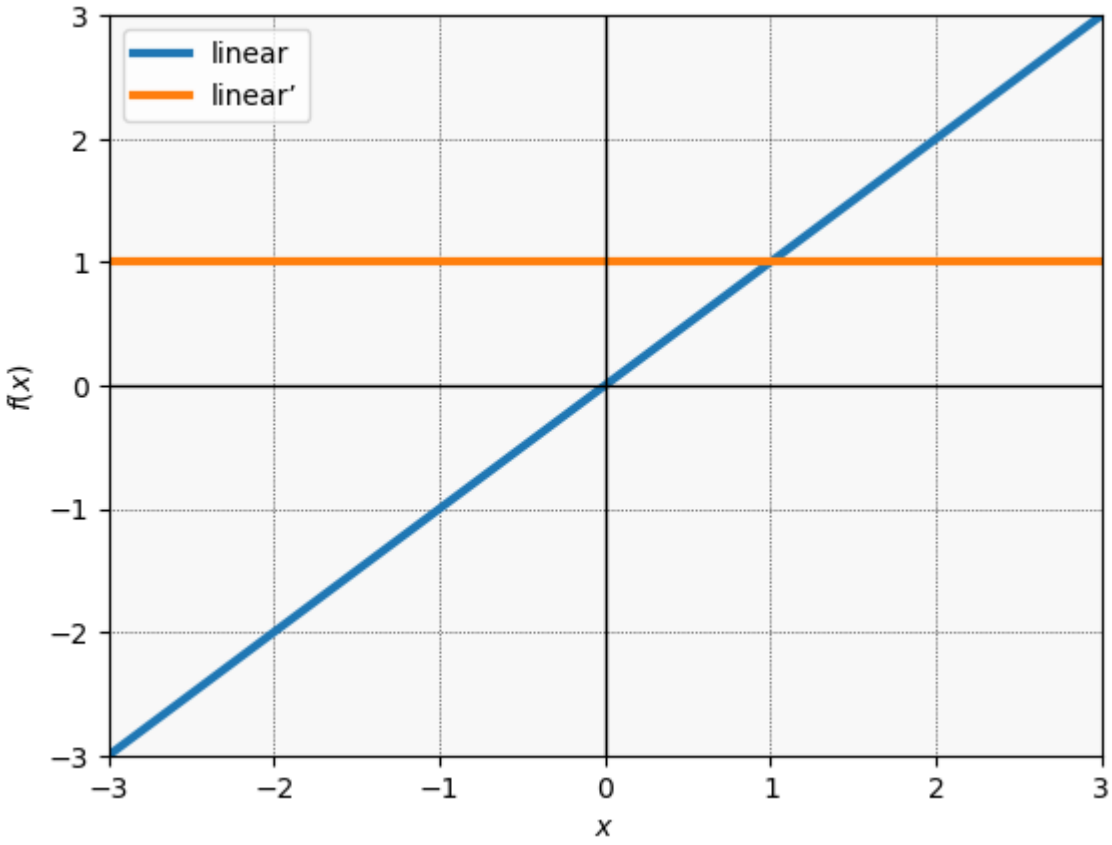
```
def derivative(function, delta=...):

    def wrapper_derivative(x):
        return ...   # Hier kun je o.a. "function(x)" aanroepen om
de afgeleide mee te berekenen

        wrapper_derivative.__name__ = function.__name__ + '''
        wrapper_derivative.__qualname__ = function.__qualname__ +
'''
    return wrapper_derivative
```

Opmerking:

De regels met `__name__` en `__qualname__` dienen slechts om de geretourneerde functie een andere naam te geven, om te voorkomen dat de afgeleide van elke willekeurige functie steeds maar `wrapper_derivative` zou blijven heten.

Controleer dat de onderstaande code correct zowel de functie als diens afgeleide weergeeft. Probeer ook jouw andere activatie-functies uit.

In [5]:
```
my_activation = activation_functions.linear
my_gradient = vlearning.derivative(my_activation)
data.graph([my_activation, my_gradient])
```

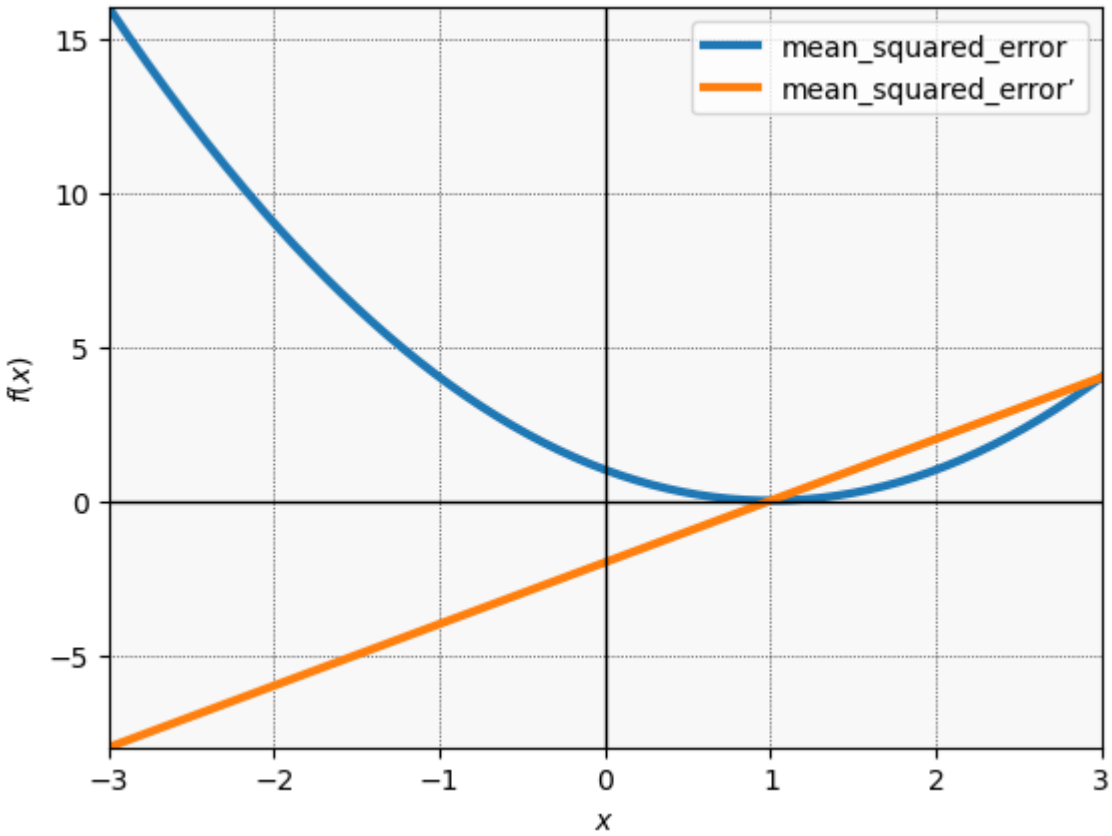


Omdat sommige functies ook nog andere parameters verwachten dient de resulterende afgeleide functie dergelijke extra parameters te kunnen verwerken. Zo dient bijvoorbeeld de loss-functie naast het argument `yhat` ook een extra argument `y` te ontvangen. Dat betekent dat ook de afgeleide functie naast het argument `yhat` die extra argumenten moet meekrijgen. Gebruik hiervoor de `*args` notatie.

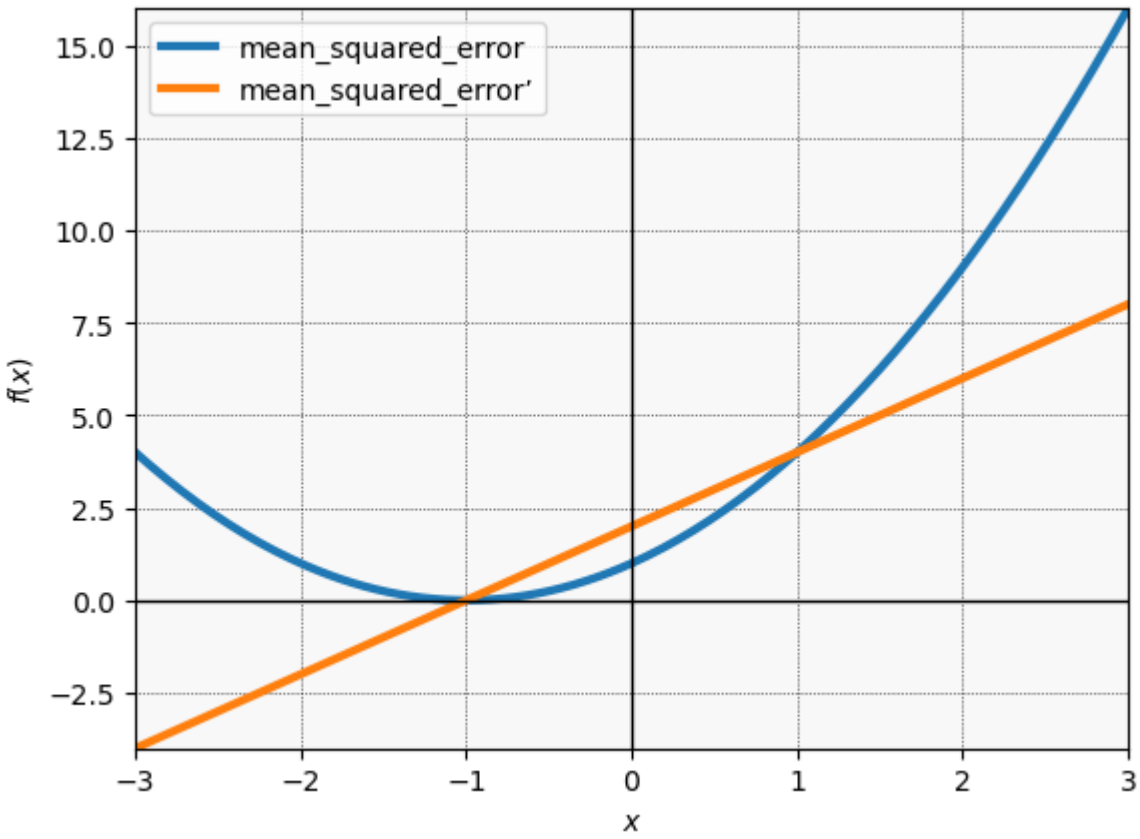
```
def wrapper_derivative(x, *args):
        return ...   # Hier kun je o.a. "function(x, *args)"
aanroepen om de afgeleide mee te berekenen
```

Verifieer dat de onderstaande code correct zowel de functie als diens afgeleide weergeeft. De loss-functie dient in dit geval dus weer te geven hoe groot de loss is voor een instance met een gegeven uitkomst y. Ga na dat de kwadratische loss-functie een minimum heeft als $\hat{y} = y$, wat betekent dat de afgeleide daar door nul gaat. Geldt dit ook voor de absolute loss-functie?

In [6]:
```
my_loss = loss_functions.mean_squared_error
my_gradient = vlearning.derivative(my_loss)
y = +1.0
data.graph([my_loss, my_gradient], y)
```



In [7]:
```
y = -1.0
data.graph([my_loss, my_gradient], y)
```



Gefeliciteerd!

Je hebt nu zelf een functie geïmplementeerd die de numerieke afgeleide van elke willekeurige functie kan berekenen.

Lineaire regressie

We gaan de implementaties van het perceptron en lineaire regressie uit het vorige hoofdstuk generaliseren. Dit algemene model zal in latere hoofdstukken gebruikt worden als uitgangspunt voor het maken van neurale netwerken. Voeg aan je bestaande module één nieuwe class genaamd `Neuron()` toe met de volgende inhoud:

```
class Neuron():

    def __repr__(self):
                text = f'Neuron(dim={self.dim}, activation=
{self.activation.__name__}, loss={self.loss.__name__})'
                return text
```

Deze class krijgt tijdens diens initialisatie een stel parameters mee die bepalen wat de te gebruiken activatie- en loss-functies zullen zijn. Door hier geschikte waarden voor mee te geven kan met één en hetzelfde model zowel classificatie als regressie worden uitgevoerd!

Laten we echter eerst weer een kijkje nemen naar de data. De functie `data.linear()` produceert wederom een verzameling willekeurige instances met bijbehorende uitkomsten. Voorlopig kiezen we voor regressie met numerieke uitkomsten middels de parameter `outcome='numeric'`.

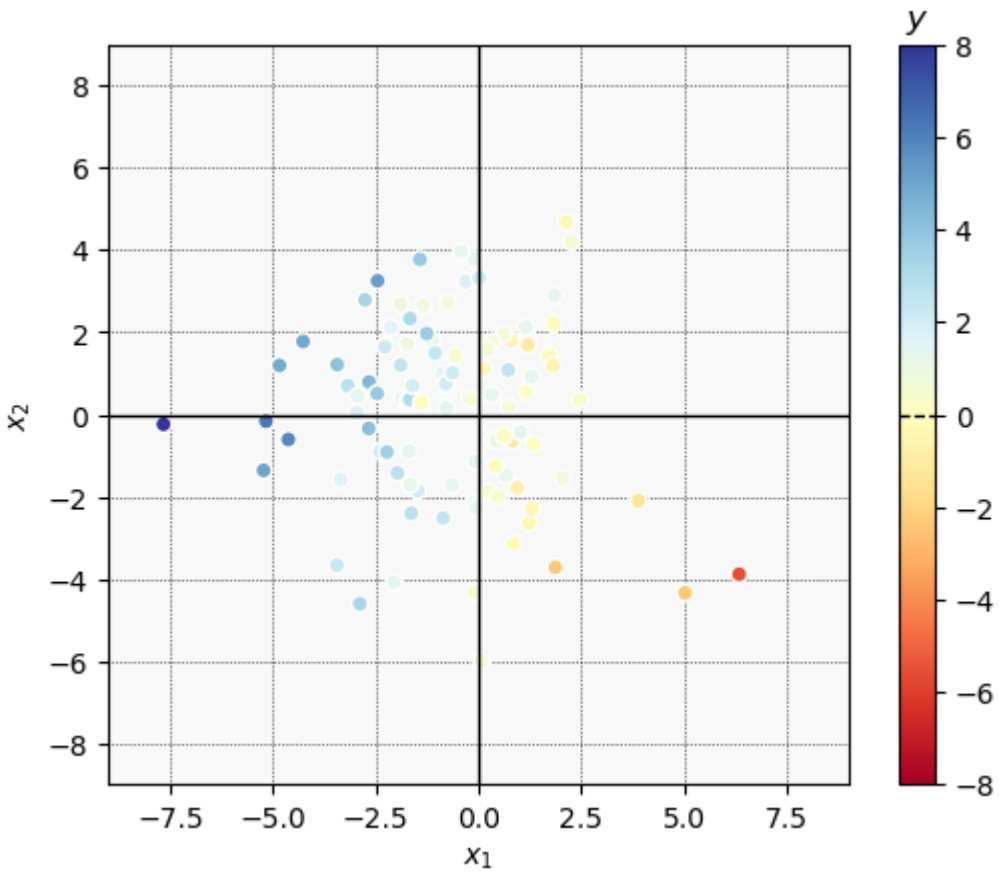
In [8]:
```
xs, ys = data.linear(outcome='numeric', noise=1.0)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys).head()
```

Out[8]:

	x1	x2	y
0	1.225973	1.848465	0.327709
1	-5.168312	-0.161938	6.301506
2	-1.126324	1.802534	1.019613
3	-3.349206	-1.581269	1.558043
4	-1.853834	0.382829	1.543387

Zoals je ziet maken we nu ook gebruik van een extra parameter genaamd `noise`. Als deze groter dan nul wordt gekozen dan zullen de datapunten willekeurig enigszins worden verplaatst, waardoor ze niet langer lineair separabel hoeven te zijn of niet meer exact een lineair model zullen volgen.

In [9]:
```
data.scatter(xs, ys)
```



Experimenteer eens met de `noise` parameter om een idee te krijgen van diens effect. Vergelijk bijvoorbeeld de instelling `noise=0.0` met `noise=10.0`; wat valt je daarbij op aan de resulterende datasets?

De initialisatie-methode `__init__()` dient als eerste te worden toegevoegd aan onze `Neuron()` class. We onderscheiden voor ons generieke model wederom de bias b en gewichten $\boldsymbol{w}$. Daarnaast zullen zoals gezegd de activatie- en loss-functies moeten worden toegekend aan instance-variabelen; geef deze als default waarden respectievelijk de functies `linear()` en `mean_squared_error()`. Bekijk de methode `__repr__()` om te zien welke variabelen in elk geval gedefinieerd dienen te zijn.

Als het goed is kun je hierna zonder foutmeldingen een nieuw object instantiëren en weergeven. Hieronder geeft de parameter `2` aan dat de dataset twee numerieke attributen omvat. Controleer dat de bias en gewichten juist zijn geinitialiseerd.

In [10]:
```
my_neuron = vlearning.Neuron(2)
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```

```
Neuron(dim=2, activation=linear, loss=mean_squared_error)
- bias = 0.0
- weights = [0.0, 0.0]
```

Probeer ook uit of het neuron met een andere gewenste activatie- of loss-functie kan worden aangemaakt.

In [11]:
```
my_neuron = vlearning.Neuron(dim=2, activation=activation_functions.sign, lo
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```

```
Neuron(dim=2, activation=sign, loss=mean_absolute_error)
- bias = 0.0
- weights = [0.0, 0.0]
```

De volgende stap is om de code te schrijven die voor een gegeven instance een voorspelling kan doen van de juiste uitkomst op grond van het model van het neuron:

$$\hat{y} = \varphi\left(b + \sum_i w_i \cdot x_i\right)$$

waarin φ de activatie-functie aangeeft. De methode `predict(self, xs)` heeft een parameter die de attributen van een lijst instances ontvangt en dient waarden te retourneren die overeenkomen met het resultaat van de bovenstaande formule. Dit is zeer vergelijkbaar met dezelfde functie van het perceptron uit het vorige hoofdstuk.

Als je deze code correct hebt geïmplementeerd zou je hieronder de voorspellingen in tabelvorm moeten zien. Het model doet nog geen zinvolle voorspellingen, dus de waarden voor $\hat{y}$ en y komen nog niet overeen.

In [12]:
```
my_neuron = vlearning.Neuron(dim=2)
yhats = my_neuron.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[12]:

	x1	x2	y	ŷ
0	1.225973	1.848465	0.327709	0.0
1	-5.168312	-0.161938	6.301506	0.0
2	-1.126324	1.802534	1.019613	0.0
3	-3.349206	-1.581269	1.558043	0.0
4	-1.853834	0.382829	1.543387	0.0

Vervolgens gaan we het neuron trainen op grond van een verzameling instances met gegeven attributen en uitkomsten. Gebruik hiervoor telkens de volgende update-regel:

$$\begin{cases} b \leftarrow b - \alpha \cdot \dfrac{\partial l}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial a} & (1) \\ w_i \leftarrow w_i - \alpha \cdot \dfrac{\partial l}{\partial \hat{y}} \cdot \dfrac{\partial \hat{y}}{\partial a} \cdot x_i & (2) \end{cases}$$

De verschillende afgeleiden kun je bepalen met de eerder gedefinieerde `derivative()` functie:

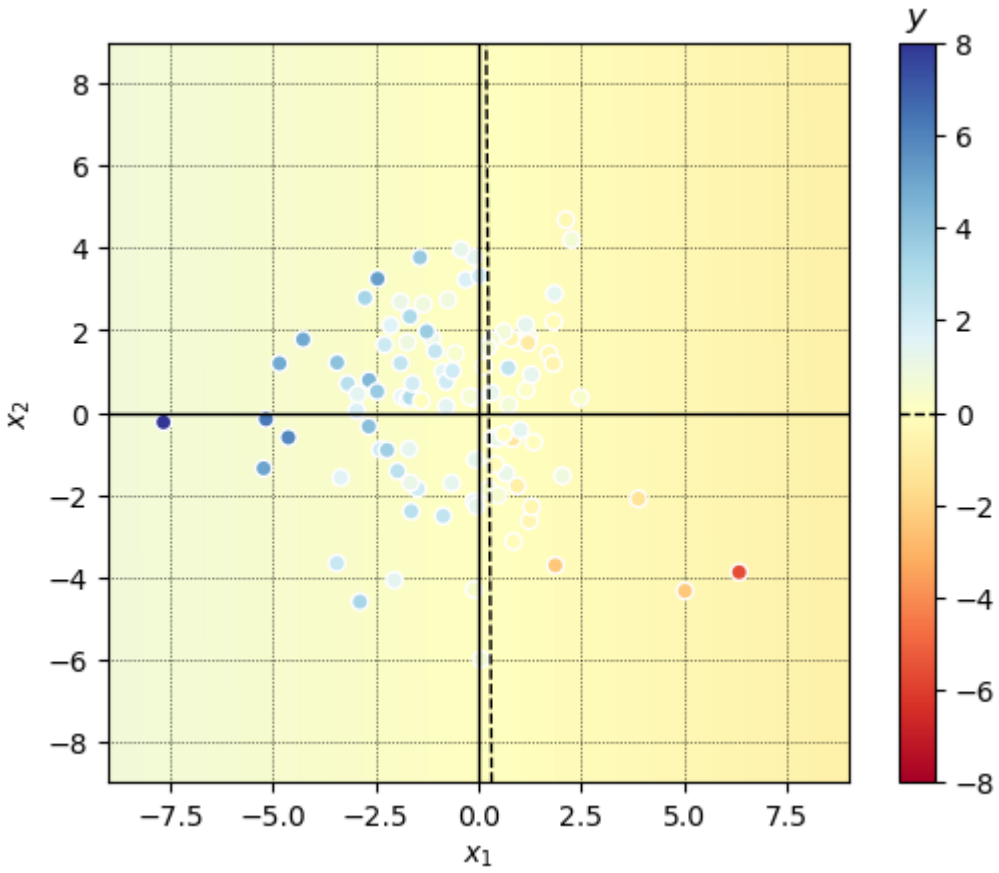
- $\frac{\partial l}{\partial \hat{y}} = \frac{\partial \mathcal{L}(\hat{y};y)}{\partial \hat{y}} = \mathcal{L}'$ bevat de afgeleide van de loss-functie $\mathcal{L}$;
- $\frac{\partial \hat{y}}{\partial a} = \frac{\partial \varphi(a)}{\partial a} = \varphi'$ bevat de afgeleide van de activatie-functie φ.

De methode `partial_fit(self, xs, ys, *, alpha=...)` heeft naast de attributen en de uitkomsten ook weer de learning rate als parameter. Vergelijk dit met je lineaire regressiemodel van het vorige hoofdstuk. Kies weer een redelijke default waarde voor α.

Na het trainen met vijf instances zou er al een zwakke gradiënt in de achtergrond zichtbaar kunnen worden. De diagonale stippellijn geeft hieronder aan waar de voorspelling $\hat{y} = 0$; deze scheidt dus de instances met een voorspelde positieve uitkomst van die met een voorspelde negatieve uitkomst. Op dit moment hoeven de voorspellingen nog niet goed met de data overeen te komen.

In [13]:
```
my_neuron = vlearning.Neuron(dim=2)
my_neuron.partial_fit(xs[:5], ys[:5])
data.scatter(xs, ys, model=my_neuron)
```

```
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```

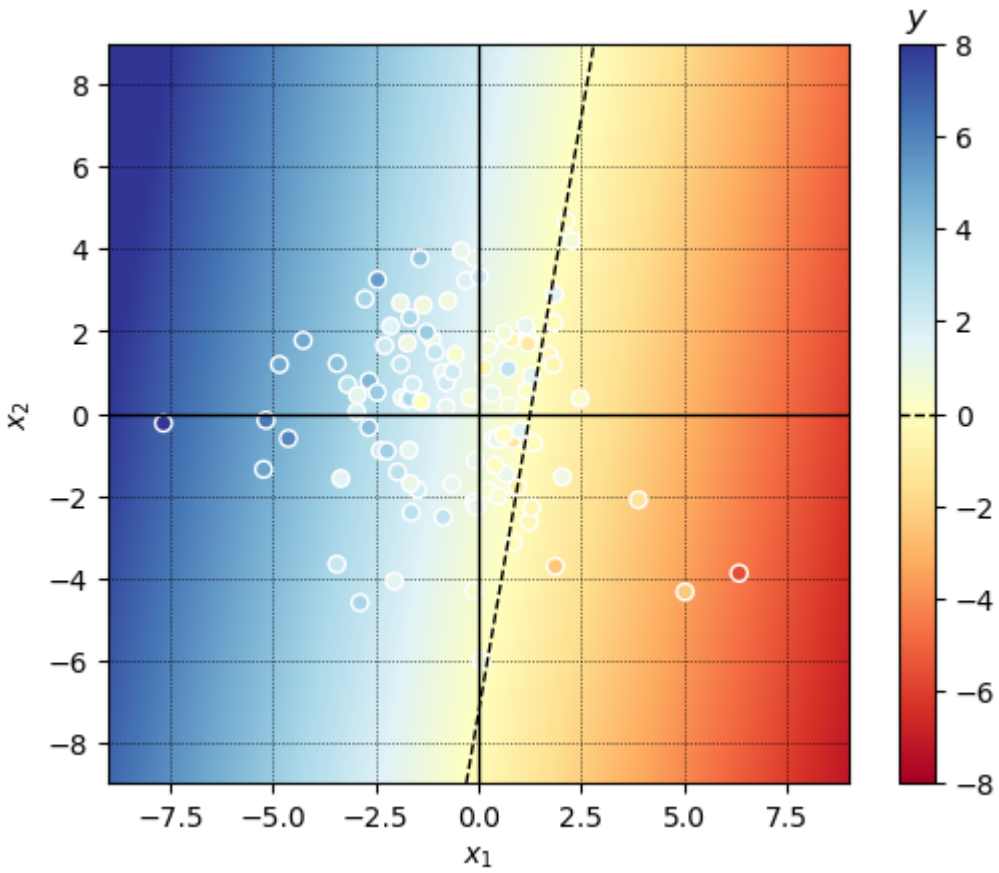



```
Neuron(dim=2, activation=linear, loss=mean_squared_error)
- bias = 0.020558150337976762
- weights = [-0.08048936454007224, -0.0005850668169808683]
```

De laatste methode die geïmplementeerd dient te worden is weer `fit(self, xs, ys, *, alpha=..., epochs=...)`. Deze dient de hele dataset te gebruiken om het neuron te trainen gedurende een gegeven aantal epochs. Vermoedelijk kun je deze methode nagenoeg letterlijk overnemen van je eerdere `LinearRegression()` class. Daar werd ook de rol van de extra keyword-argumenten beschreven.

Draai de code hieronder. Slaagt je model erin om te convergeren naar een uitkomst die de echte getalwaarden van de instances ogenschijnlijk goed voorspelt?

In [14]:
```
my_neuron = vlearning.Neuron(dim=2)
my_neuron.fit(xs, ys)
data.scatter(xs, ys, model=my_neuron)
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```




```
Neuron(dim=2, activation=linear, loss=mean_squared_error)
- bias = 0.9693553756944173
- weights = [-0.7709232166629649, 0.1330734211046624]
```

Verifieer hieronder dat de voorspellingen enigszins overeen komen met de gewenste waarden. Omdat de data nu ruis bevatten is de overeenkomst niet exact, maar voor instances waarvoor de uitkomsten ver van nul vandaan liggen zou het teken van y en $\hat{y}$ toch overeen moeten komen.

In [15]:
```
yhats = my_neuron.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[15]:

	x1	x2	y	ŷ
0	1.225973	1.848465	0.327709	0.270206
1	-5.168312	-0.161938	6.301506	4.932178
2	-1.126324	1.802534	1.019613	2.077534
3	-3.349206	-1.581269	1.558043	3.340911
4	-1.853834	0.382829	1.543387	2.449463

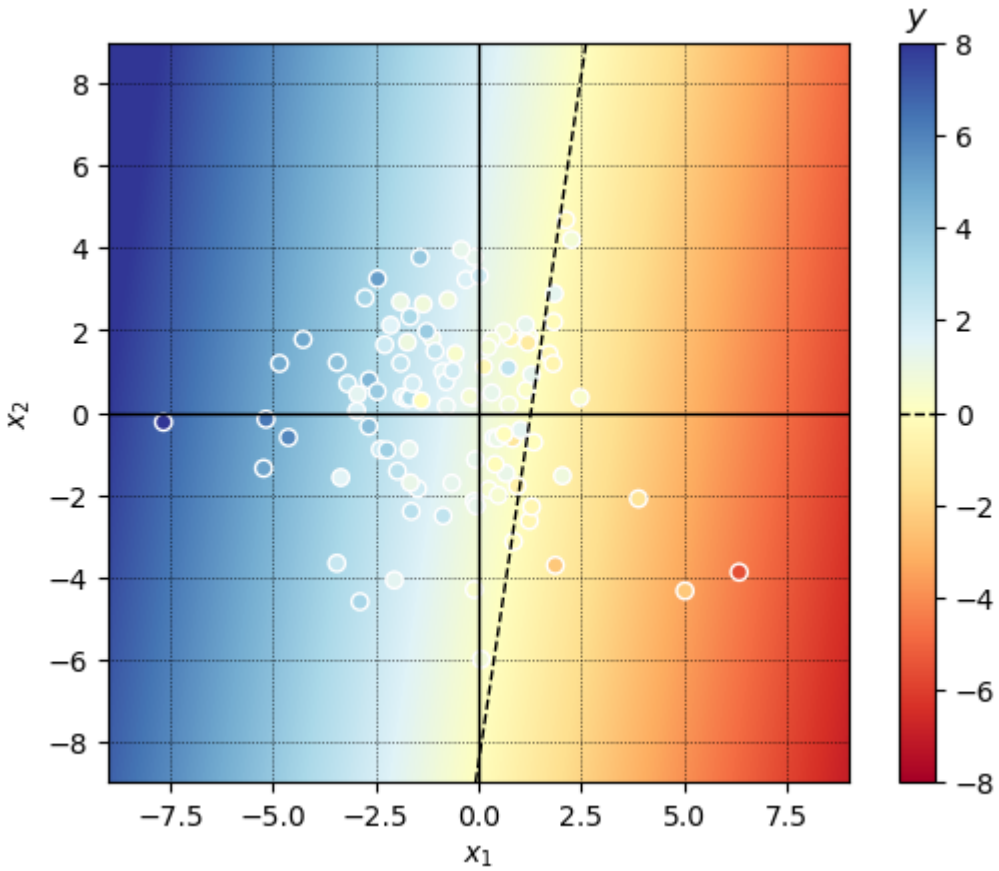
Gefeliciteerd!

Je hebt nu een algoritme geïmplementeerd dat modellen met willekeurige activatie- en loss-functies kan oplossen.

Ter vergelijking passen we hieronder ook weer het lineaire regressiemodel uit de machine-learning module scikit-learn op deze dataset met ruis toe.

Ziet de oplossing er hetzelfde uit als voor je eigen model? Zijn de waarden van de bias en de gewichten van vergelijkbare grootten?

In [16]:
```
skl_linearregression = linear_model.LinearRegression()
skl_linearregression.fit(xs, ys)
data.scatter(xs, ys, model=skl_linearregression)
print(skl_linearregression)
print(f'- bias = {skl_linearregression.intercept_}')
print(f'- weights = {skl_linearregression.coef_}')
```




```
LinearRegression()
- bias = 0.9758074968221484
- weights = [-0.76475082  0.11426635]
```

Logistische regressie

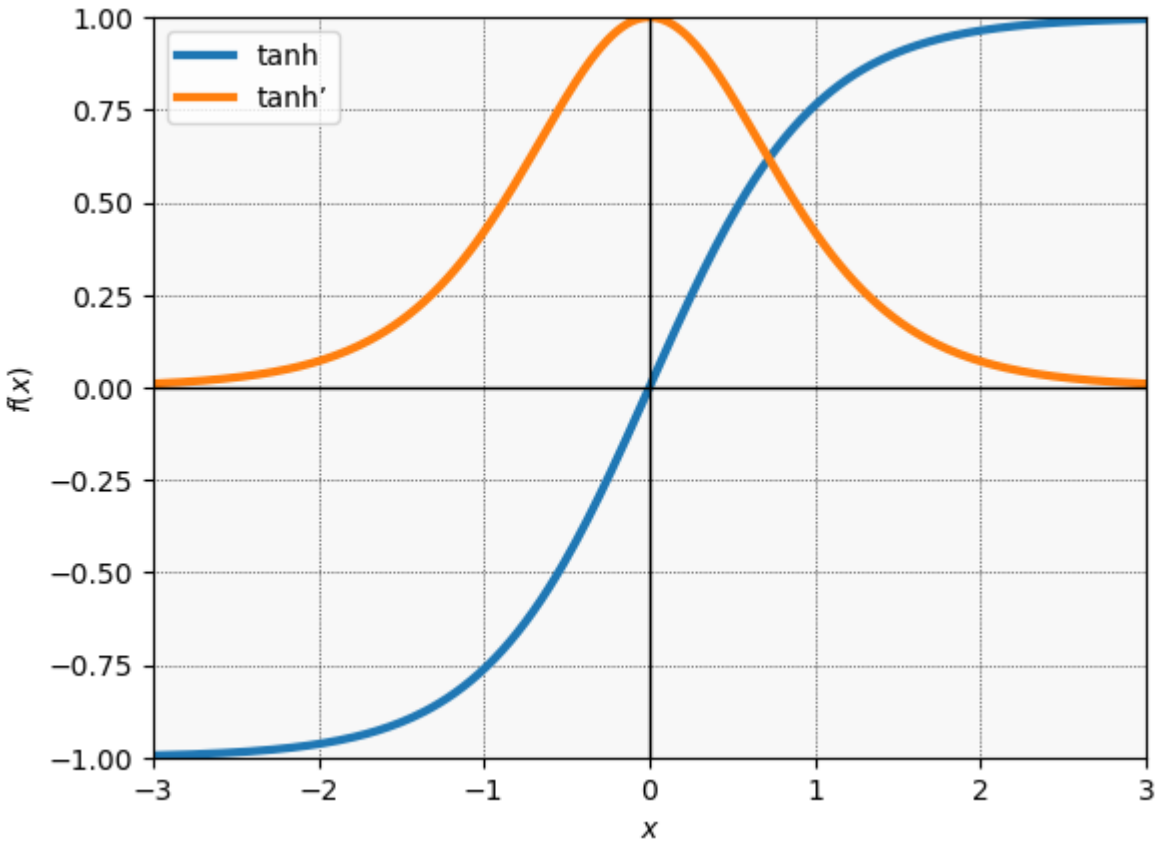
Als het goed is kan je met dit model ook classificatie uitvoeren in plaats van regressie. We hoeven daartoe alleen nog maar de activatie-functie te veranderen, en "klaar is Kees (m/v/x)".

Ogenschijnlijk kun je het perceptron implementeren door de signum-functie als activatiefunctie te kiezen, maar helaas heeft deze geen geschikte afgeleide (de helling is altijd ofwel nul, ofwel oneindig groot). Dit kan opgelost worden door een afgevlakte sigmoïde functie te nemen. Een voorbeeld hiervan is de tanh-functie. Deze lost zowel het probleem van helling nul als van helling oneindig op, aangezien deze overal een positieve maar eindige helling heeft.

> **Opmerking:**
>
> Eigenlijk maakt logistische regressie ook nog gebruik van een andere loss-functie, de *cross-entropy*. We maken hier in latere hoofdstukken kennis mee, maar voorlopig gebruiken we gewoon de kwadratische loss-functie.

Implementeer de `tanh()` functie en plot deze functie hieronder samen met zijn afgeleide. Python's `math` module kan hier behulpzaam zijn. Het resultaat van het gebruik van een dergelijke activatie-functie heet logistische regressie.

In [17]:
```
my_activation = activation_functions.tanh
my_gradient = vlearning.derivative(my_activation)
data.graph([my_activation, my_gradient])
```

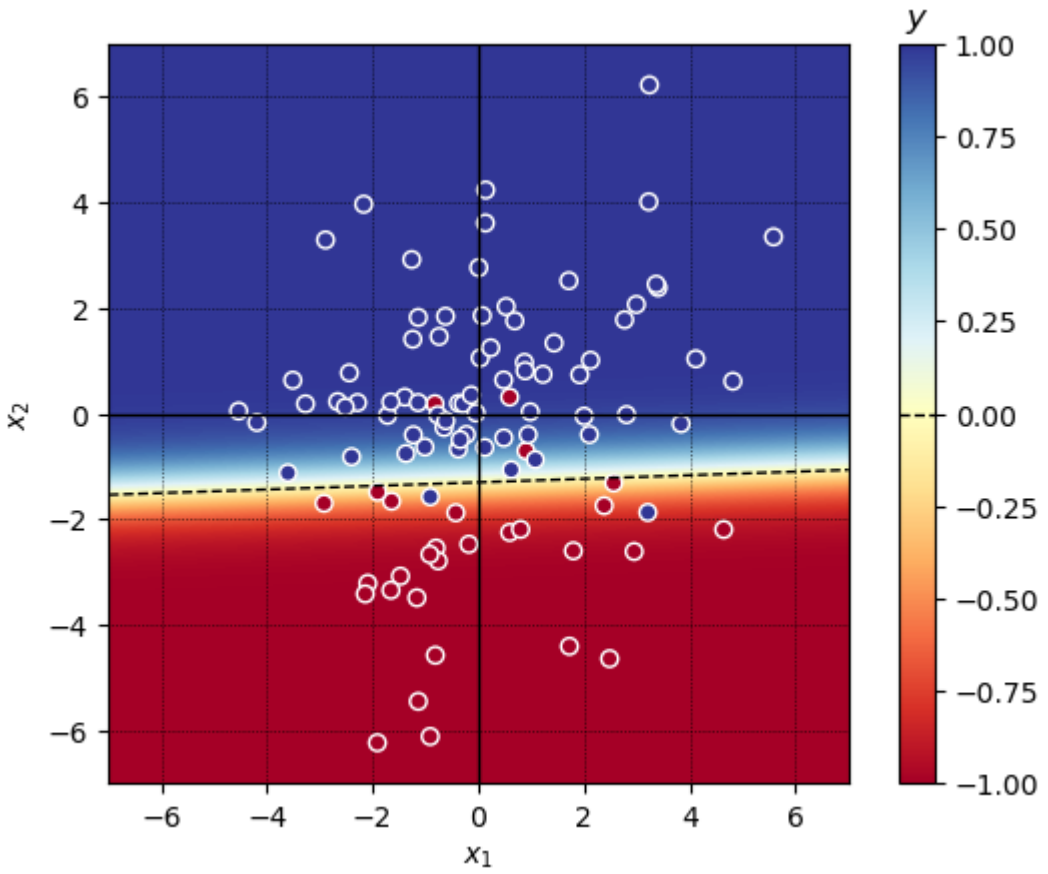


Hieronder wordt een dataset met nominale klasselabels gefit met deze activatiefunctie. De overgang tussen de klassen is door de afgevlakte tanh-functie niet zo scherp als bij het perceptron, maar voor data die niet lineair separabel zijn is dit juist zinvol omdat

hiermee aangegeven kan worden dat de classificatie nabij de grenslijn enigszins onzeker is.

Kijk of de code correct functioneert. Is de gevonden scheidingslijn voor zover mogelijk in staat om de instances van de beide klassen te onderscheiden?

In [18]:
```
xs, ys = data.linear(outcome='nominal', noise=1.0)
my_neuron = vlearning.Neuron(dim=2, activation=activation_functions.tanh)
my_neuron.fit(xs, ys)
data.scatter(xs, ys, model=my_neuron)
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```




```
Neuron(dim=2, activation=tanh, loss=mean_squared_error)
- bias = 1.935816817091496
- weights = [-0.05006477277789124, 1.4870514779951514]
```

In tabelvorm ziet het er als volgt uit. De voorspelde $\hat{y}$ hebben niet de waarden -1 en +1, maar iets ertussenin afhankelijk van hoe (on)zeker het model is over de classificatie.

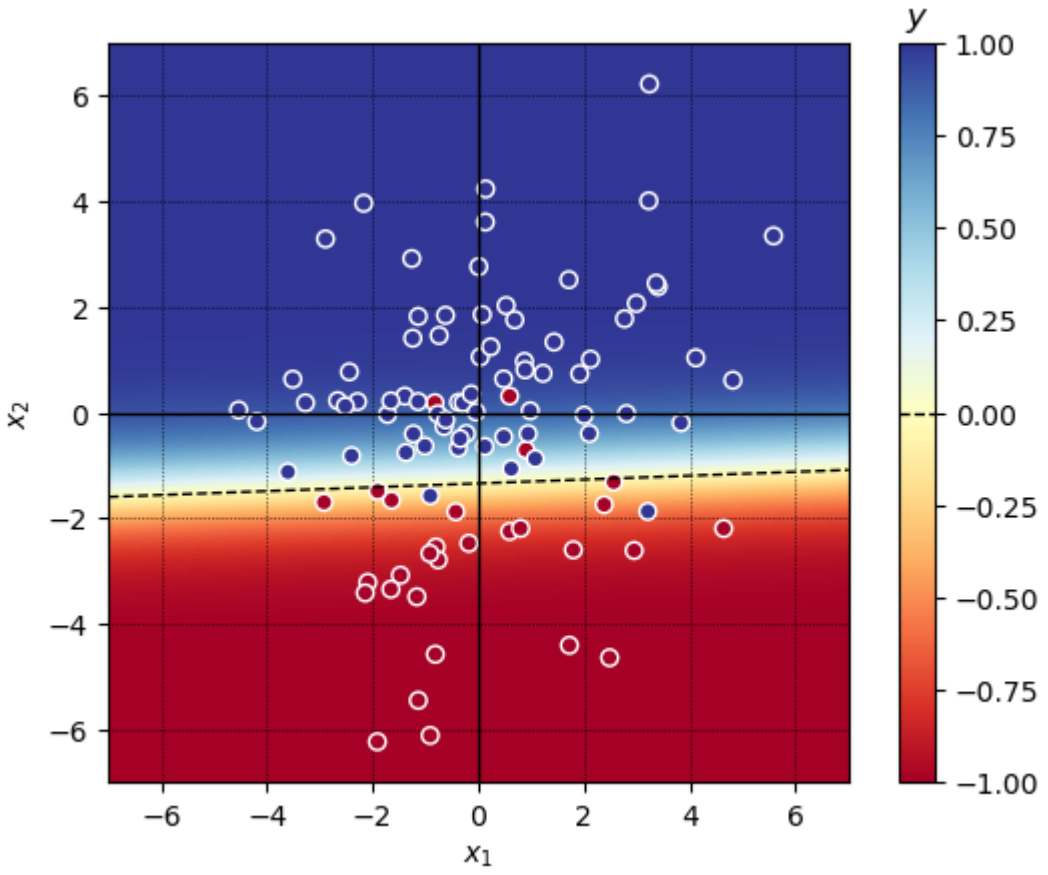
In [19]:
```
yhats = my_neuron.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[19]:

	x1	x2	y	ŷ
0	2.121630	1.007798	1.0	0.997432
1	0.899847	-0.705680	-1.0	0.686541
2	0.033788	1.051911	1.0	0.998172
3	-3.511935	0.629963	1.0	0.995510
4	4.815053	0.610921	1.0	0.989098

Ook dit resultaat vergelijken we hieronder weer met het logistische regressiemodel uit de machine-learning module scikit-learn. Ziet de oplossing er enigszins hetzelfde uit als voor je eigen model?

In [20]:
```
skl_logisticregression = linear_model.LogisticRegression()
skl_logisticregression.fit(xs, ys)
data.scatter(xs, ys, model=skl_logisticregression)
print(skl_logisticregression)
print(f'- bias = {skl_logisticregression.intercept_[0]}')
print(f'- weights = {skl_logisticregression.coef_[0]}')
```




```
LogisticRegression()
- bias = 2.570136524444818
- weights = [-0.06991574  1.91660403]
```

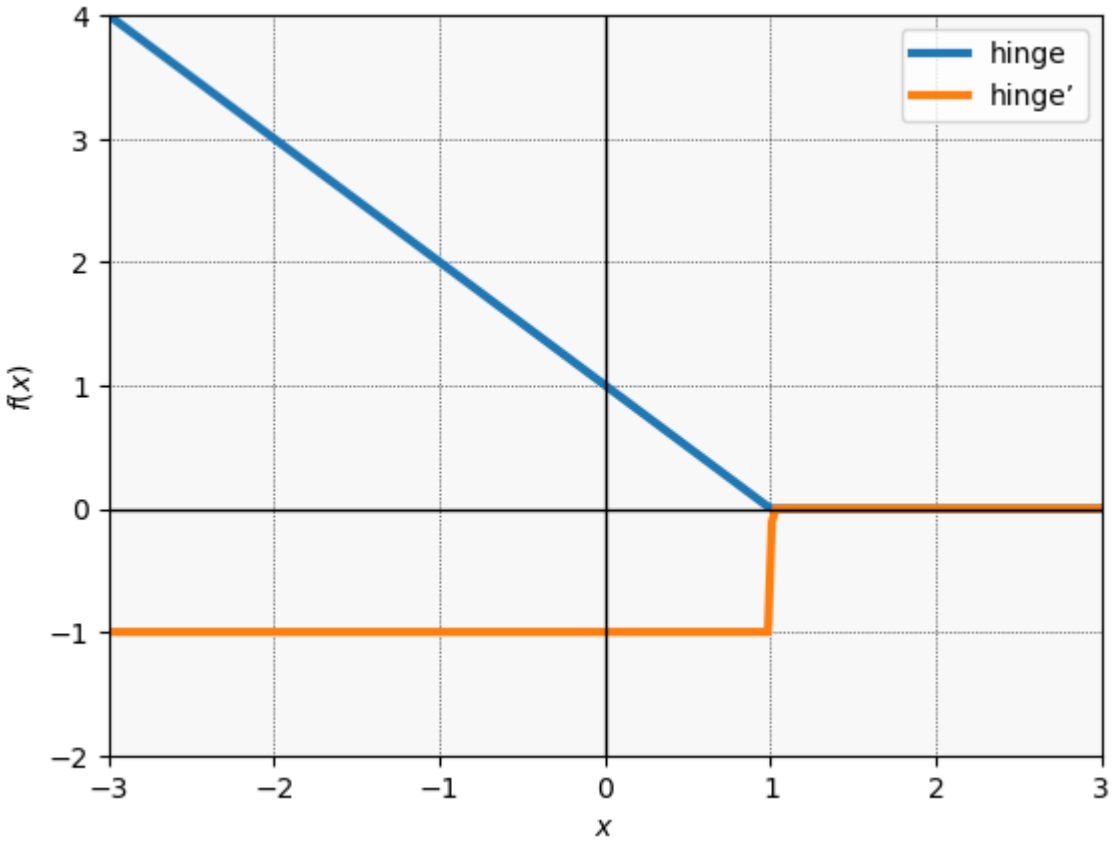
Support Vector Machines

Om de kracht van onze ontwikkelde `Neuron()` class te demonstreren sluiten we af met een algoritme dat we kunnen benaderen door niet een andere activatie-functie maar een andere loss-functie te kiezen. Een lineaire Support Vector Machine (SVM) kan worden geformuleerd in termen van een zogenaamde hinge loss-functie $\mathcal{L}\left(\hat{y}; y\right) = \max\left(1 - \hat{y} \cdot y, 0\right)$. Voor de activatie-functie wordt nu de identiteitsfunctie genomen, zoals bij lineaire regressie.

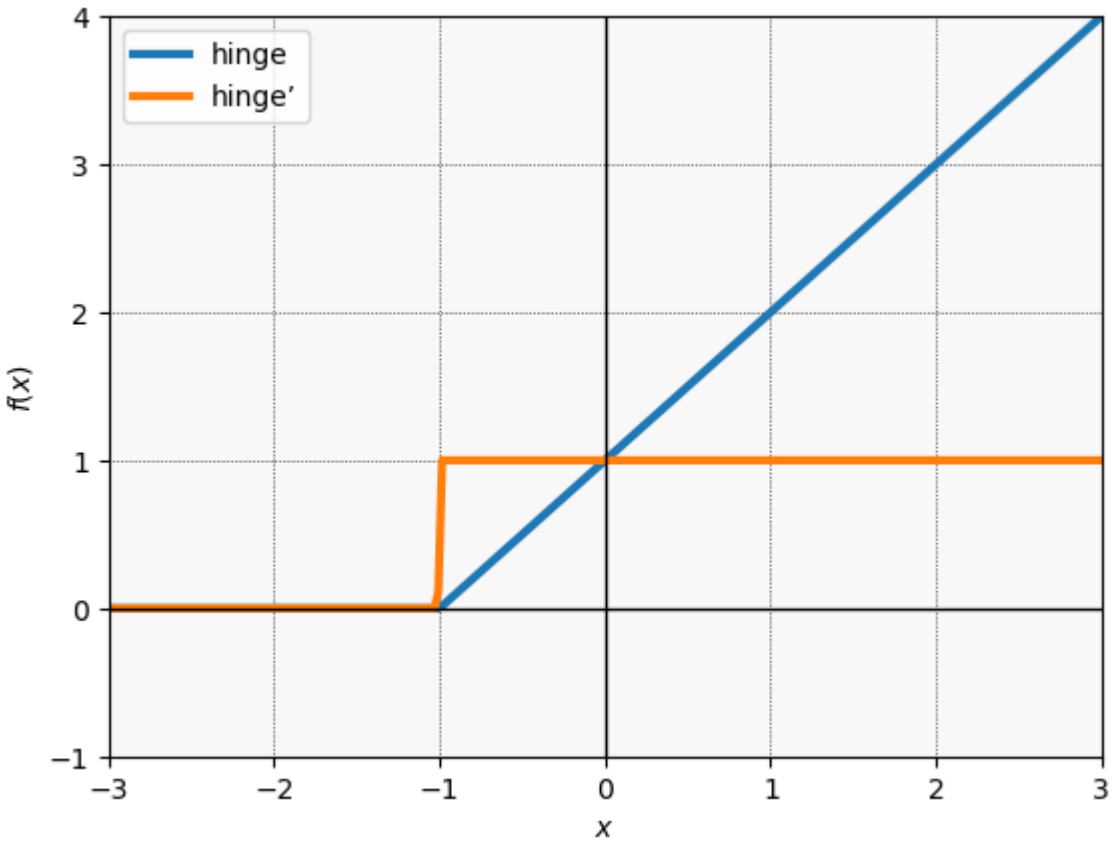
> **Opmerking:**
>
> Net als logistische regressie zit een SVM eigenlijk iets ingewikkelder in elkaar dan we hier pretenderen. Deze maakt namelijk tevens gebruik van *regularisatie*. Ook daar komen we in een later hoofdstuk nog op terug.

Voeg de hierboven genoemde functie `hinge()` aan je model toe en plot deze hieronder.

In [21]:
```
my_loss = loss_functions.hinge
my_gradient = vlearning.derivative(my_loss)
y = +1.0
data.graph([my_loss, my_gradient], y)
```



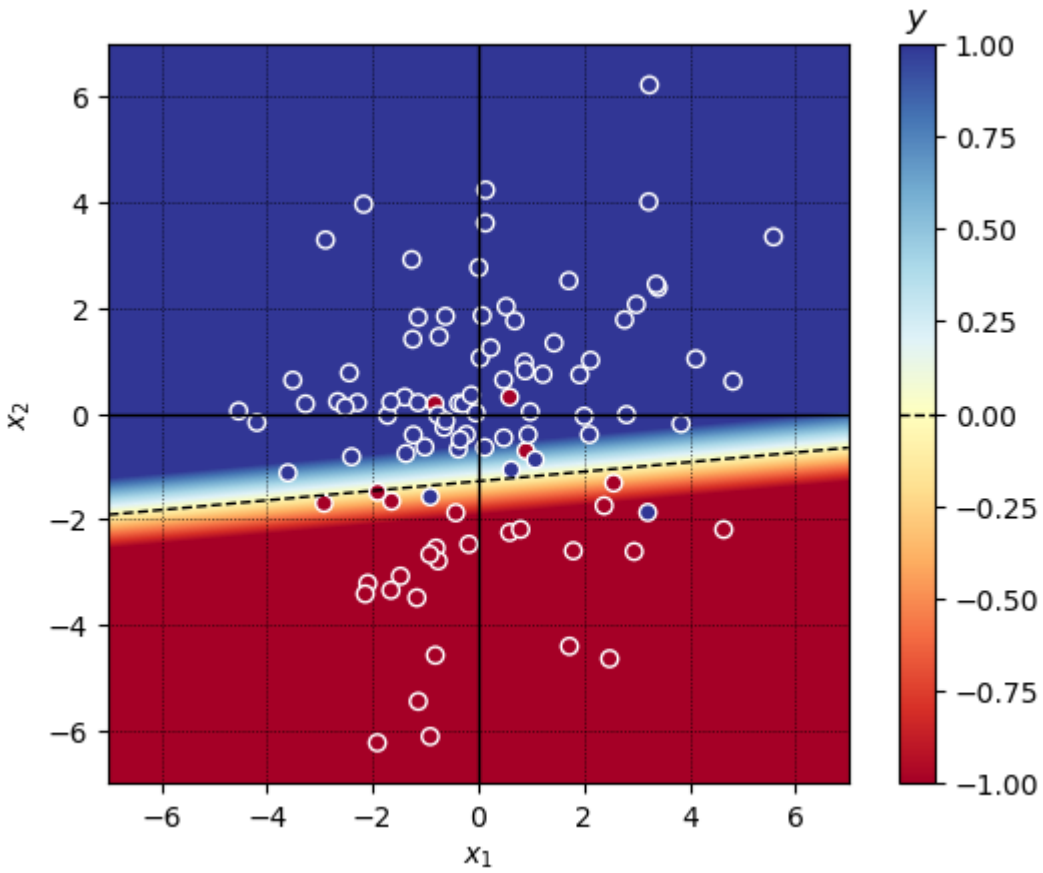
In [22]:
```
y = -1.0
data.graph([my_loss, my_gradient], y)
```



We zullen deze functie niet uitgebreid bespreken, maar te zien is wel dat de hinge-functie de instances die verkeerd geclassificeerd worden een toenemende cost toekent naarmate ze verder aan de verkeerde kant van de scheidingsmarge liggen. Dat wil zeggen, instances met $y = +1$ worden toenemend bestraft als $\hat{y}$ (langs de x-as in de grafiek) negatief is en instances met $y = -1$ worden toenemend bestraft als $\hat{y}$ positief is. Als instances aan de goede kant liggen wordt er begrijpelijkerwijs een loss gelijk aan nul toegekend. In een overgangsgebied $-1 < \hat{y} < +1$ krijgen álle instances een kleine penalty.

Hoe dan ook, draai de onderstaande code om de eerdere data te classificeren. Geeft je SVM ook met deze activatie- en loss-functies een op het oog redelijke scheidingslijn aan?

In [23]:
```
my_neuron = vlearning.Neuron(dim=2, loss=loss_functions.hinge)
my_neuron.fit(xs, ys)
data.scatter(xs, ys, model=my_neuron)
print(my_neuron)
print(f'- bias = {my_neuron.bias}')
print(f'- weights = {my_neuron.weights}')
```




```
Neuron(dim=2, activation=linear, loss=hinge)
- bias = 1.8911758521740558
- weights = [-0.1336751090940279, 1.477265904479195]
```

Voor de volledigheid kijken we ook hier naar de bijbehorende getalwaarden. Omdat de SVM geen sigmoïde activatiefunctie bevat die de uitkomsten beperkt tot een bereik van -1 tot +1, duiden in dit geval positieve getallen $\hat{y} > 0$ op een klasselabel +1 en negatieve getallen $\hat{y} < 0$ op een klasselabel -1. Het teken van $\hat{y}$ zou dus idealiter moeten overeenkomen met dat van y.

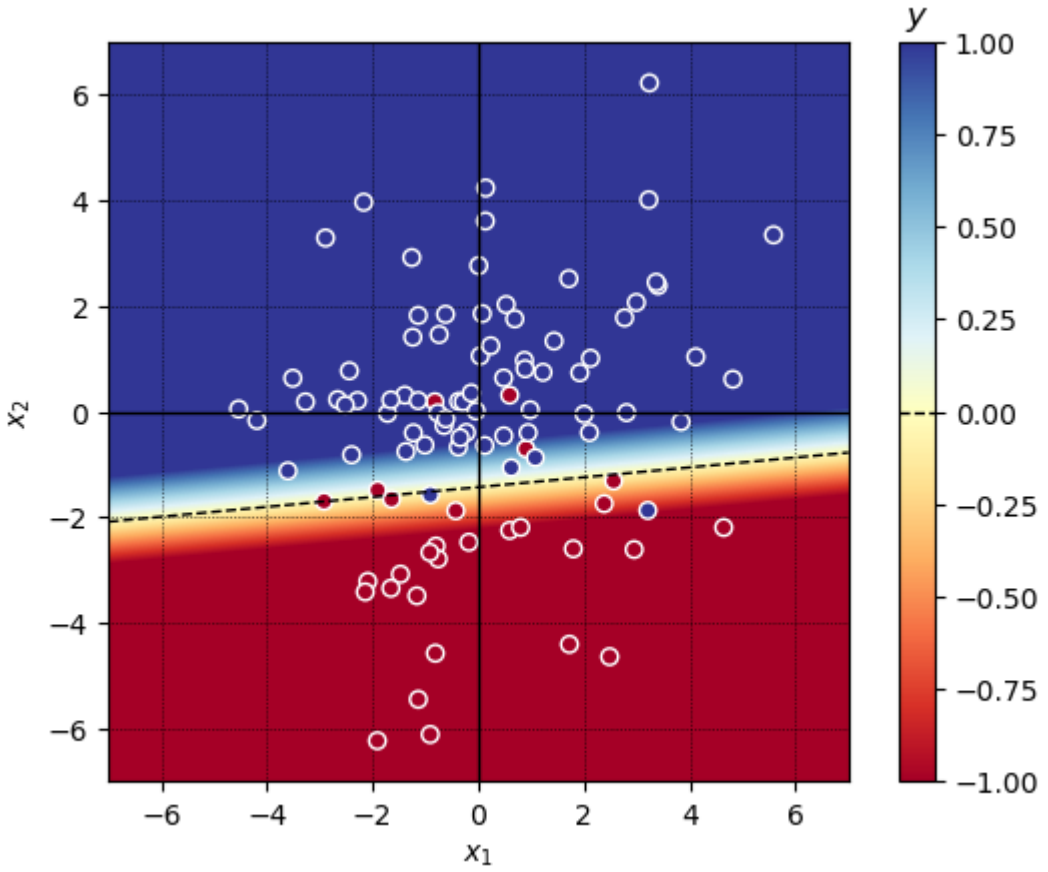
In [24]:
```
yhats = my_neuron.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[24]:

	x1	x2	y	ŷ
0	2.121630	1.007798	1.0	3.096352
1	0.899847	-0.705680	-1.0	0.728412
2	0.033788	1.051911	1.0	3.440611
3	-3.511935	0.629963	1.0	3.291257
4	4.815053	0.610921	1.0	2.150015

Ook dit resultaat vergelijken we hieronder weer met een SVM uit de machine-learning module scikit-learn. Ziet de oplossing er vergelijkbaar uit als voor je eigen model?

In [25]:
```
skl_svm = svm.SVC(kernel='linear')
skl_svm.fit(xs, ys)
data.scatter(xs, ys, model=skl_svm)
print(skl_svm)
print(f'- bias = {skl_svm.intercept_[0]}')
print(f'- weights = {skl_svm.coef_[0]}')
```




```
SVC(kernel='linear')
- bias = 1.7108837507598456
- weights = [-0.11166725  1.19610373]
```

Gefeliciteerd!

Je hebt nu zelf een uitermate flexibel, multifunctioneel algoritme ontwikkeld dat een perceptron, lineaire regressie, logistische regressie, SVMs (en nog veel meer machine-learning algoritmen) kan uitvoeren.
