Deel I. Machine learning

Hoofdstuk 1. Het perceptron

Inleiding

Dit is het Jupyter Notebook behorende bij hoofdstuk 1 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. Creeër daarnaast een bestand `model.py` in een code-editor naar keuze en sla al deze bestanden gezamenlijk op in één werkmap.

> **Opmerking:**
> Op BlackBoard tref je ook een `*.html` versie van dit notebook aan met daarin de evaluaties van alle onderstaande code. Gebruik deze zonodig als voorbeeld om te zien welke uitvoer zoal gewenst is.

We beginnen met het importeren van de functies in de `data` module, samen met de nu nog lege `model` module die je zelf gaat schrijven. Verder laden we alvast enkele ondersteunende functies uit de modules `sklearn` en `pandas` die we later zullen gebruiken, en voeren we wat initialisatie-code uit. Plaats de cursor in de cel hieronder en druk op Ctrl+Enter (of Shift+Enter om meteen naar de volgende cel te gaan).

> **Opmerking:**
> Als het label in de linkermarge verandert van `In [ ]:` via `In [*]:` naar `In [1]:` is de code succesvol uitgevoerd en werkt dit notebook naar behoren.

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, __version__
print(f'Using sklearn version {__version__}')
```

```
import model, data
```

```
Using python version 3.11.0
Using pandas version 1.5.3
Using sklearn version 1.2.1
```

> **Opmerking:**
> Als `pandas` of `sklearn` niet geinstalleerd is op je systeem, voer dan `pip3 install pandas sklearn` uit in een terminal en herstart de python kernel via de menu-optie `Kernel` > `Restart` van dit notebook.

Laten we eerst eens beter kijken naar de dataset die we in dit werkcollege gaan gebruiken. De functie `data.linear()` produceert een verzameling willekeurige instances met numerieke attributen. Er zijn onder andere parameters om het aantal instances en het aantal attributen te bepalen. Al deze parameters hebben geschikte default waarden. Een verplichte string parameter geeft aan of de dataset discrete uitkomsten dient te hebben t.b.v. classificatie (`outcome='nominal'`) of continue uitkomsten t.b.v. regressie (`outcome='numeric'`).

In [2]:
```
help(data.linear)
```

```
Help on function linear in module data:

linear(outcome, *, num=100, dim=2, noise=0.0, seed=None)
    Generate a linear dataset with attributes and outcomes.

    Arguments:
    outcome  -- string indicating 'nominal' or 'numeric' outcomes

    Keyword options:
    num      -- number of instances (default 100)
    dim      -- dimensionality of the attributes (default 2)
    noise    -- the amount of noise to add (default 0.0)
    seed     -- a seed to initialise the random numbers (default random)

    Return values:
    xs       -- values of the attributes
    ys       -- values of the outcomes
```

De functie retourneert een geneste lijst met vectoren die de attributen van de instances bevat en een enkelvoudige lijst met klasselabels danwel getalwaarden die de bijbehorende uitkomsten bevat. De nominale data zijn lineair separabel; de continue data volgen een exacte lineaire relatie. De algoritmen die we in dit werkcollege implementeren zouden hierdoor in principe in staat behoren te zijn deze data *perfect* te modelleren.

Het perceptron

Laten we beginnen met een dataset met twee attributen (x_1, x_2) en met nominale klasselabels y waarop we classificatie kunnen toepassen d.m.v. het perceptron. De eerste vijf instances worden hieronder weergegeven in tabelvorm.

In [3]:
```
xs, ys = data.linear('nominal')
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys).head()
```

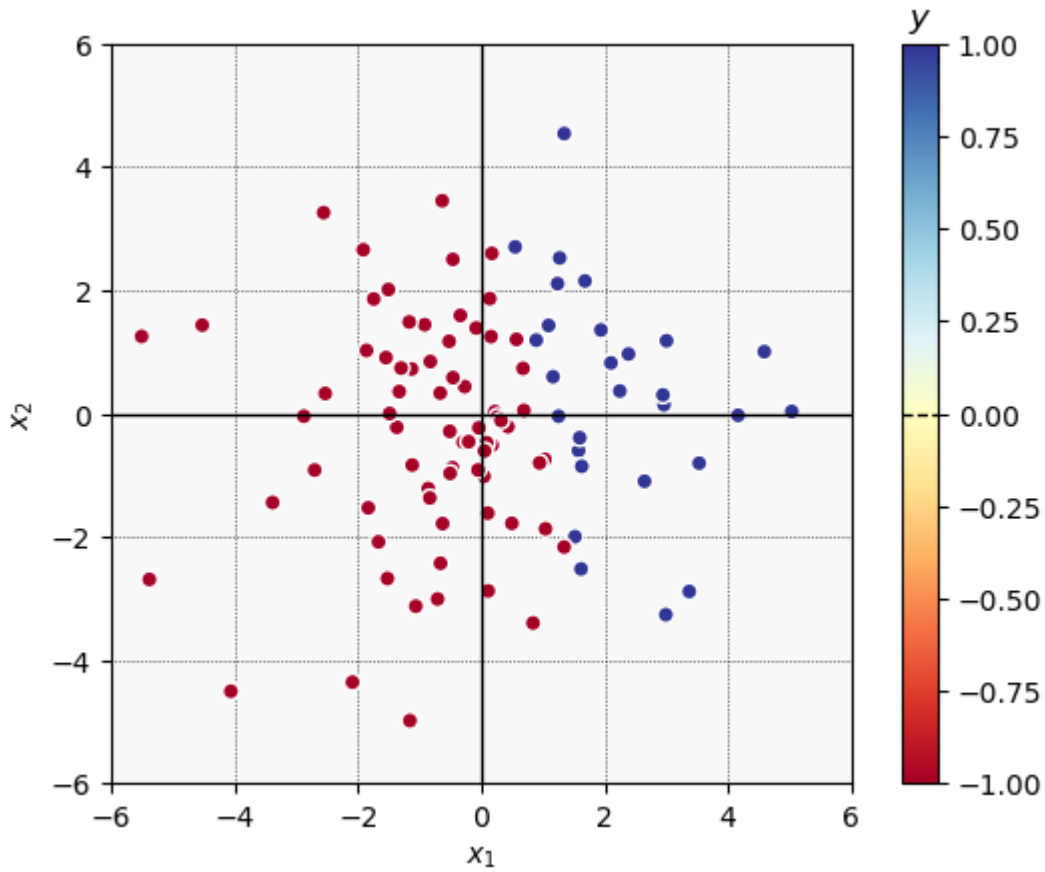
Out[3]:

	x1	x2	y
0	-4.528164	1.432952	-1.0
1	1.679769	2.149753	1.0
2	-2.532304	0.324853	-1.0
3	-0.667908	0.331646	-1.0
4	1.575954	-0.600291	1.0

Verken zelf nader de structuur van de `xs` en `ys` variabelen: wat voor type variabelen zijn het, hoe lang zijn de lijsten, en zo verder.

De functie `data.scatter()` geeft de instances van beide klassen weer middels een kleurcode (rood voor de klasse $y = -1$ en blauw voor de klasse $y = +1$).

In [4]:
```
data.scatter(xs, ys)
```



Definieer een `Perceptron()` class in het bestand `model.py` met aanvankelijk alleen de onderstaande `__repr__()` methode die een textuele representatie geeft van het object:

```
class Perceptron():
1
    def __repr__(self):
        text = f'Perceptron(dim={self.dim})'
        return text
```

> **Opmerking:**
> Vergeet niet om tijdens het uitwerken van deze oefening je aanpassingen in de editor telkens op te slaan voordat je de code in dit notebook uitvoert.

We beginnen met het implementeren van de initialisatie-methode `__init__()`. Deze krijgt één verplichte parameter `dim` die aangeeft hoeveel attributen de te classificeren instances zullen hebben. Verder dienen er twee instance-variabelen te worden geïnitialiseerd: `bias` met de bias b en `weights` met de gewichten $\boldsymbol{w}$; de methode dient deze zelf van geschikte beginwaarden te voorzien.

Het resultaat is iets als:

```
def __init__(self, dim):
        self.dim = ...
        self.bias = ...
        self.weights = ...
```

> **Opmerking:**
> Voorzie je eigen code waar nodig van documentatie en commentaren, evenals desgewenst assertions en foutafhandeling. In de voorbeelden in deze notebooks worden deze niet getoond omwille van bondigheid, maar het gebruik hiervan wordt desalniettemin aangeraden.

Als het goed is kun je nu zonder foutmeldingen een nieuw eigen `Perceptron()` object instantiëren en weergeven met de volgende code.

In [5]:
```
my_perceptron = model.Perceptron(dim=2)
print(my_perceptron)
print(f'- bias = {my_perceptron.bias}')
print(f'- weights = {my_perceptron.weights}')
```

```
Perceptron(dim=2)
- 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 het juiste klasselabel op grond van het model van het perceptron:

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

Creeër een methode `predict(self, xs)` met een parameter `xs` die de attributen van een lijst instances ontvangt. Deze methode dient een lijst waarden te retourneren die overeenkomen met de uitkomsten van de bovenstaande formule. De invoer van deze functie is dus een geneste lijst van lijsten; de uitvoer is een enkelvoudige lijst.

Als je deze code correct hebt geïmplementeerd kunnen we het model vragen om een voorspelling te doen omtrent de labels van de beschikbare data. Omdat het perceptron nog niet getraind is is de voorspelling $\hat{y}$ overal gelijk aan nul. Als je hieronder geen foutmeldingen krijgt dan functioneert je predictie-methode vooralsnog.

In [6]:
```
my_perceptron = model.Perceptron(dim=2)
yhats = my_perceptron.predict(xs)
my_perceptron.partial_fit(xs[:5], ys[:5])
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[6]:

	x1	x2	y	ŷ
0	-4.528164	1.432952	-1.0	0.0
1	1.679769	2.149753	1.0	0.0
2	-2.532304	0.324853	-1.0	0.0
3	-0.667908	0.331646	-1.0	0.0
4	1.575954	-0.600291	1.0	0.0

Vervolgens gaan we het perceptron trainen op grond van instances met gegeven attributen en klasselabels. Pas hiervoor met de aangeleverde instances één voor één de update-regel toe:

$$\begin{cases} b \leftarrow b - (\hat{y} - y) \\ w_i \leftarrow w_i - (\hat{y} - y)\, x_i \end{cases}$$

Voeg een methode `partial_fit()` toe aan je perceptron met parameters die de attributen en klasselabels van een aantal trainingsinstances ontvangt. Het resultaat is iets als:

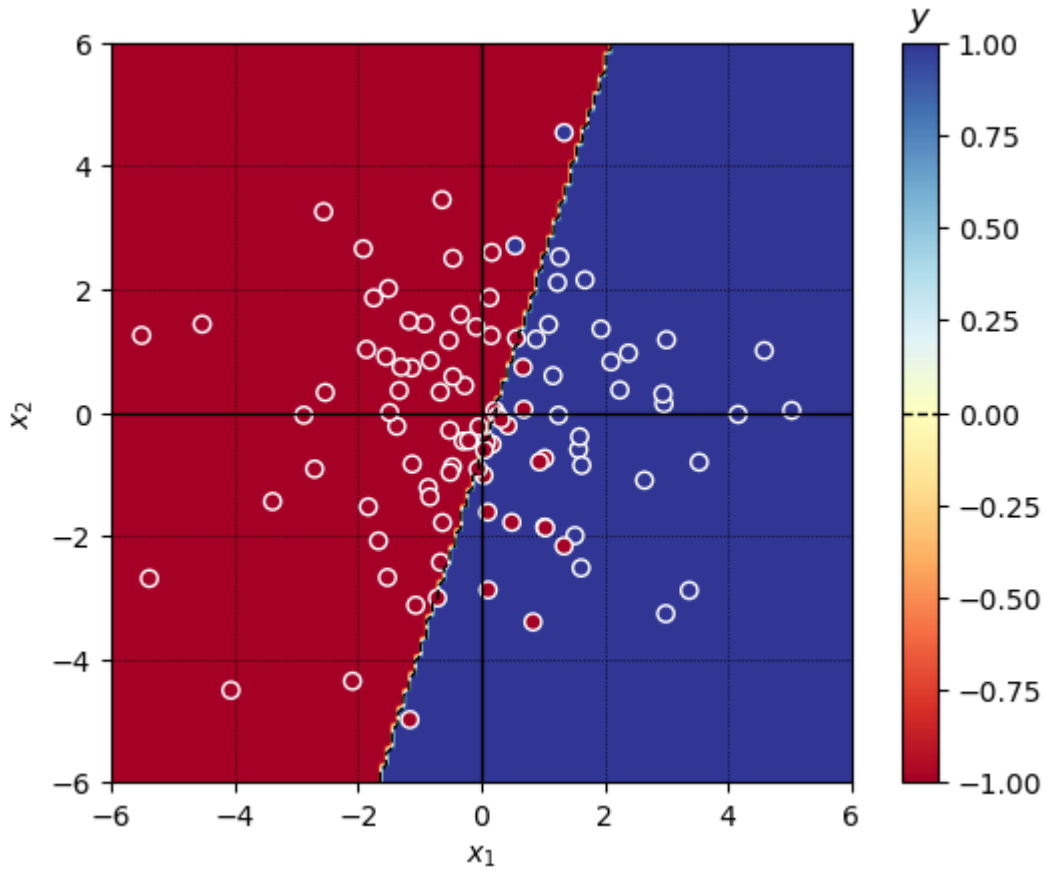
```
def partial_fit(self, xs, ys):
        for x, y in zip(xs, ys):
            ...   # Update hier het perceptron met één instance {x, y}
```

De methode dient elke instance één maal te gebruiken om een update uit te voeren; er wordt dus één epoch getraind met de gegeven data. Deze functie hoeft niets te retourneren.

Als je deze code correct hebt geïmplementeerd zou je hieronder een gekleurde achtergrond moeten zien die weergeeft hoe het perceptron de verschillende waarden

van de attributen zou classificeren nadat het éénmaal getraind is op de eerste vijf instances uit de dataset.

In [7]:
```
my_perceptron = model.Perceptron(dim=2)
my_perceptron.partial_fit(xs[:5], ys[:5])
data.scatter(xs, ys, model=my_perceptron)
print(my_perceptron)
print(f'- bias = {my_perceptron.bias}')
print(f'- weights = {my_perceptron.weights}')
```



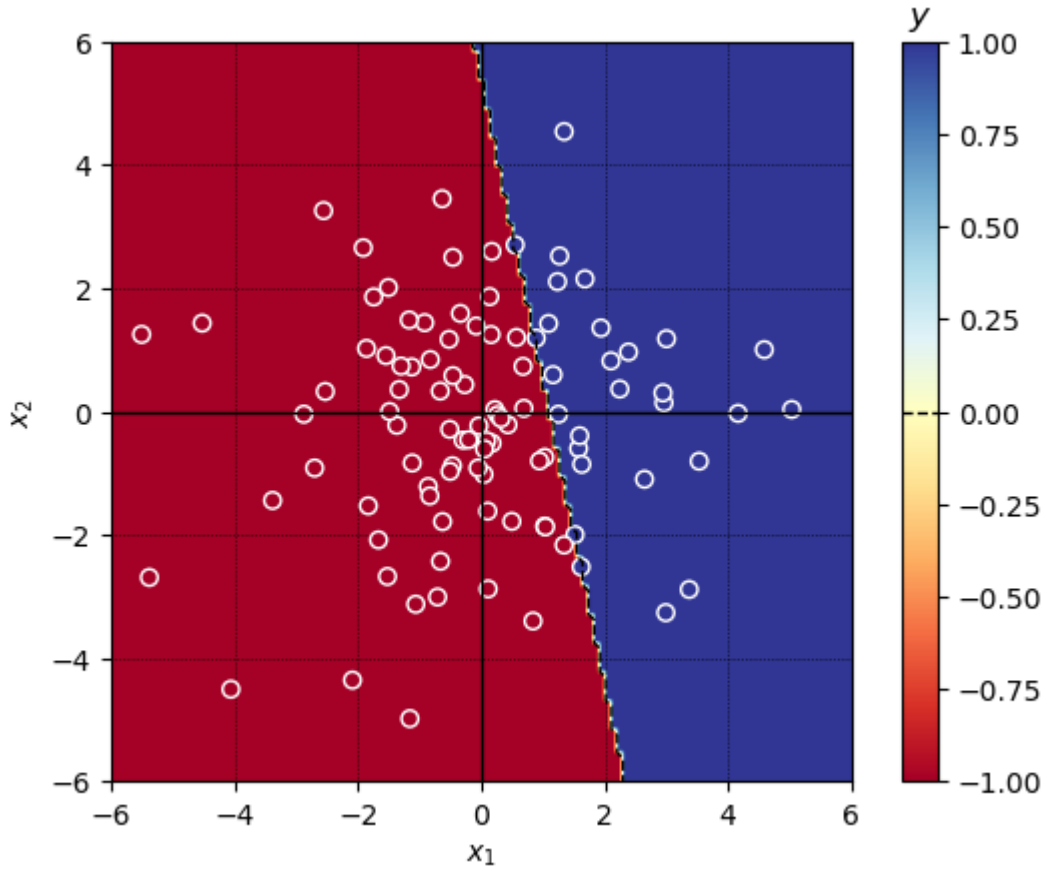

```
Perceptron(dim=2)
- bias = -1.0
- weights = [4.528164288345444, -1.4329518068048805]
```

Hoogstwaarschijnlijk zal er nog geen correcte grenslijn gevonden worden, maar het perceptron doet wel al een aantal juiste voorspellingen en je zou moeten kunnen zien dat de bias en gewichten zijn bijgewerkt.

Voeg tenslotte een methode `fit(self, xs, ys, *, epochs=0)` toe die meerdere epochs uitvoert, zoals opgegeven door de gebruiker. Maak hierbij gebruik van de eerdere `partial_fit()` functie om telkens één hele epoch te trainen. Zorg dat het perceptron automatisch stopt met het draaien van verdere epochs als er in de vorige epoch geen veranderingen in het model meer zijn aangebracht. Als de gebruiker om nul epochs verzoekt (of geen waarde meegeeft waardoor de default waarde `epochs=0` geldt), laat dan het algoritme zoveel epochs draaien als maar nodig zijn om te convergeren.

In [8]:
```
my_perceptron = model.Perceptron(dim=2)
my_perceptron.fit(xs, ys, epochs=6)
data.scatter(xs, ys, model=my_perceptron)
print(my_perceptron)
print(f'- bias = {my_perceptron.bias}')
print(f'- weights = {my_perceptron.weights}')
```

```
Model has been fully fitted after 5 epochs
```




```
Perceptron(dim=2)
- bias = -15.0
- weights = [13.648978332568582, 2.792973702399119]
```

Rosenblatt heeft bewezen dat het perceptron algoritme gegarandeerd in een eindig aantal stappen convergeert naar een oplossing die alle instances juist classificeert als de data lineair separabel zijn. Dat is hier het geval. Als je dit juist implementeert zou de bovenstaande code daarom een lijn moeten vinden die de beide klassen perfect van elkaar weet te scheiden.

Hieronder worden de eerste vijf instances in een tabel getoond; ga na dat de voorspelling $\hat{y}$ telkens gelijk is aan het echte klasselabel y.

In [9]:
```
yhats = my_perceptron.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[9]:

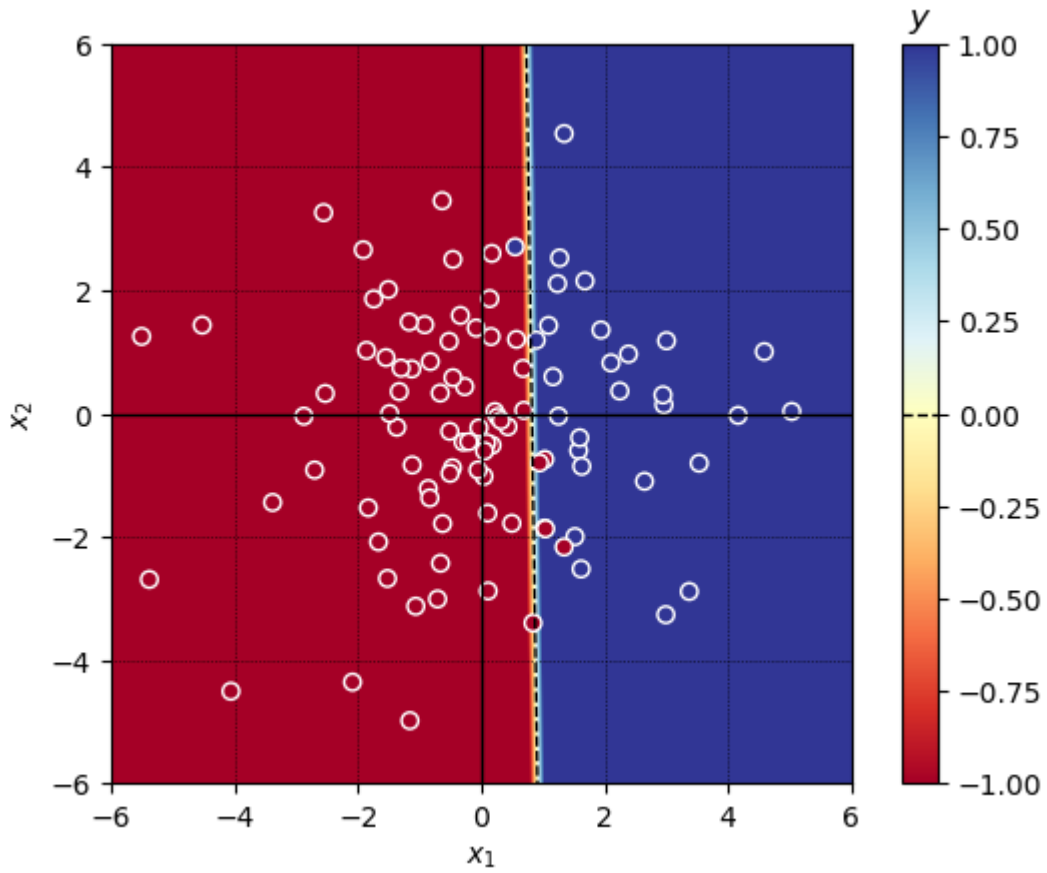
	x1	x2	y	ŷ
0	-4.528164	1.432952	-1.0	-1.0
1	1.679769	2.149753	1.0	1.0
2	-2.532304	0.324853	-1.0	-1.0
3	-0.667908	0.331646	-1.0	-1.0
4	1.575954	-0.600291	1.0	1.0

Gefeliciteerd!
Je hebt nu zelf een algoritme geïmplementeerd dat een lineair separabele verzameling trainingsdata perfect kan leren onderscheiden.

Ter vergelijking tonen we hieronder het perceptron zoals dit in de machine-learning module scikit-learn reeds beschikbaar is. De syntax van deze class is zeer vergelijkbaar met de onze, hoewel de bias en coefficienten beschikbaar zijn in de `intercept_` en `coef_` variabelen, en een parameter `max_iter` het gewenste (maximum) aantal epochs aangeeft.

Ziet de oplossing er hetzelfde uit als voor je eigen model? Ga na dat de waarden van de bias en de gewichten niet van dezelfde grootten zijn. Begrijp je hoe dit toch ogenschijnlijk hetzelfde model kan opleveren?

In [11]:
```
skl_perceptron = linear_model.Perceptron(max_iter=1000)
skl_perceptron.fit(xs, ys)
data.scatter(xs, ys, model=skl_perceptron)
print(skl_perceptron)
print(f'- bias = {skl_perceptron.intercept_[0]}')
print(f'- weights = {skl_perceptron.coef_[0]}')
```




```
Perceptron()
- bias = -7.0
- weights = [8.54603258 0.12880581]
```

Lineaire regressie

Vervolgens doen we hetzelfde met instances die een getalwaarde als te voorspellen uitkomst hebben. Eerst maar weer eens een kijkje nemen naar de data. De uitkomsten y zijn nu numeriek.

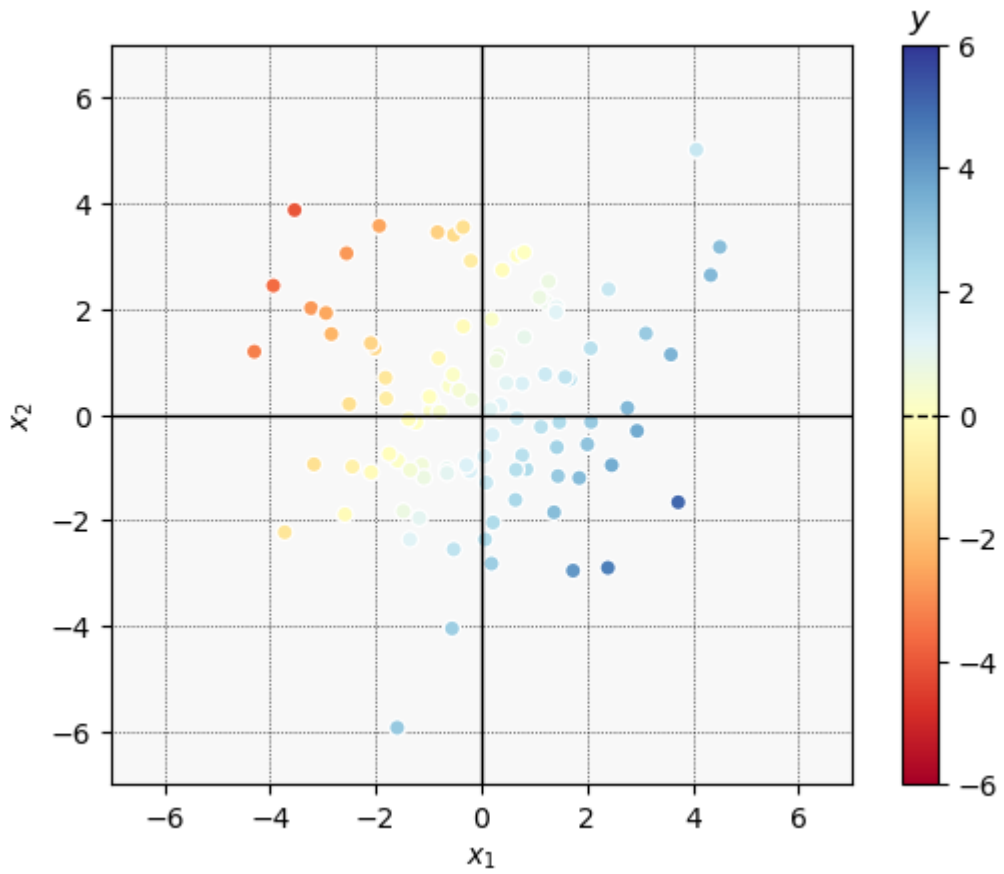
In [12]:
```
xs, ys = data.linear('numeric')
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys).head()
```

Out[12]:

	x1	x2	y
0	4.338429	2.637605	3.242041
1	-0.653553	-1.027194	1.000647
2	-0.937904	0.227610	0.086669
3	-0.215532	-1.069881	1.393017
4	3.723618	-1.659275	5.032026

De instances worden nu gekleurd langs het hele bereik van de kleurschaal. De ligging van de verschillende kleuren in het scatterplot vormt een zichtbare geleidelijke overgang van rood via geel naar blauw.

In [13]:
```
data.scatter(xs, ys)
```



Kopieer de code van het perceptron om een nieuwe class `LinearRegression()` toe te voegen aan hetzelfde bestand `model.py` en pas de initialisatie-, representatie- en predictie-methoden aan. Het model voor lineaire regressie luidt:

$$\hat{y} = b + \sum_i w_i \cdot x_i$$

De methode `predict(self, xs)` dient in dit geval een lijst met getalwaarden te retourneren i.p.v. een lijst met klasselabels.

Een ongetraind model zou wederom uitkomsten gelijk aan nul moeten voorspellen. Controleer dat je geen foutmeldingen krijgt.

In [14]:
```
my_linearregression = model.LinearRegression(dim=2)
yhats = my_linearregression.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[14]:

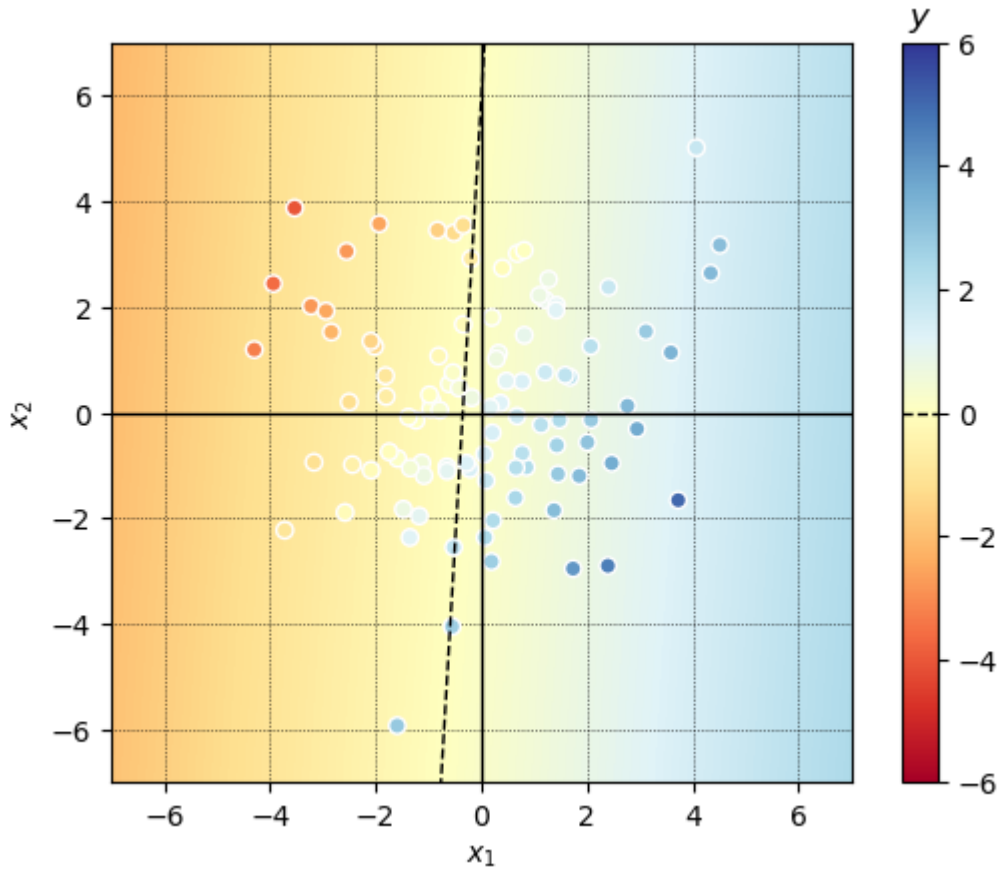
	x1	x2	y	ŷ
0	4.338429	2.637605	3.242041	0.0
1	-0.653553	-1.027194	1.000647	0.0
2	-0.937904	0.227610	0.086669	0.0
3	-0.215532	-1.069881	1.393017	0.0
4	3.723618	-1.659275	5.032026	0.0

Vervolgens gaan we het lineaire regressiemodel trainen, eerst weer op grond van de eerste vijf instances met de update-regel:

$$\begin{cases} b \leftarrow b - \alpha\,(\hat{y} - y) \\ w_i \leftarrow w_i - \alpha\,(\hat{y} - y)\,x_i \end{cases}$$

De methode `partial_fit(self, xs, ys, *, alpha=...)` krijgt nu een extra parameter, de learning rate α. Geef deze een geschikte default waarde.

In [15]:
```
my_linearregression = model.LinearRegression(dim=2)
my_linearregression.partial_fit(xs[:5], ys[:5])
data.scatter(xs, ys, model=my_linearregression)
print(my_linearregression)
print(f'- bias = {my_linearregression.bias}')
print(f'- weights = {my_linearregression.weights}')
```



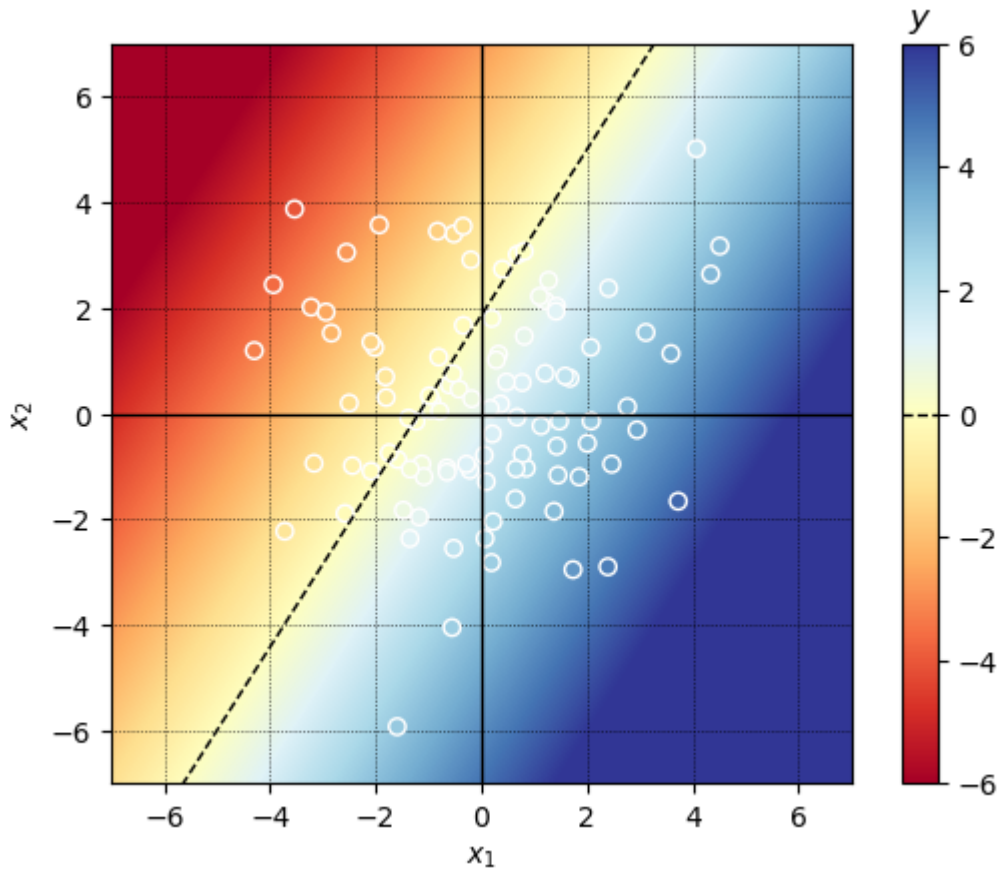
```
LinearRegression(dim=2)
- bias = 0.1058641039546127
- weights = [0.2995234195820039, -0.017670942020912753]
```

Na het trainen met enkele instances zou er al een zwakke gradiënt in de achtergrond zichtbaar kunnen worden. De diagonale stippellijn geeft hierboven aan waar de voorspelling $\hat{y} = 0$; deze scheidt dus de instances met een voorspelde positieve uitkomst van die met een voorspelde negatieve uitkomst.

Voeg nu tenslotte weer de methode `fit(self, xs, ys, *, alpha=..., epochs=...)` toe die training toepast op een gegeven aantal hele epochs. Pas de default waarde voor de learning rate weer aan, in overeenstemming met de functie `partial_fit()`. In tegenstelling tot bij het perceptron kan het aantal epochs nu niet default op nul worden gesteld, met als betekenis dat het algoritme door dient te itereren tot alle instances juist geclassificeerd zijn. Immers, het lineaire regressie model convergeert meestal niet in een eindig aantal stappen naar een exacte uitkomst. Wel wordt er bij een juiste keuze van α geleidelijk een steeds betere benadering gevonden. Vandaar dat we hier standaard een redelijk groot aantal epochs willen uitvoeren. Kies wederom zelf een geschikte default waarde.

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

In [16]:
```
my_linearregression = model.LinearRegression(dim=2)
my_linearregression.fit(xs, ys)
data.scatter(xs, ys, model=my_linearregression)
print(my_linearregression)
print(f'- bias = {my_linearregression.bias}')
print(f'- weights = {my_linearregression.weights}')
```




```
LinearRegression(dim=2)
- bias = 0.9999999999999968
- weights = [0.8434187655087624, -0.537256722608172]
```

Controleer hieronder dat de voorspellingen inderdaad goed overeen komen met de gewenste waarden.

In [17]:
```
yhats = my_linearregression.predict(xs)
DataFrame(xs, columns=['x1', 'x2']).assign(y=ys, ŷ=yhats).head()
```

Out[17]:

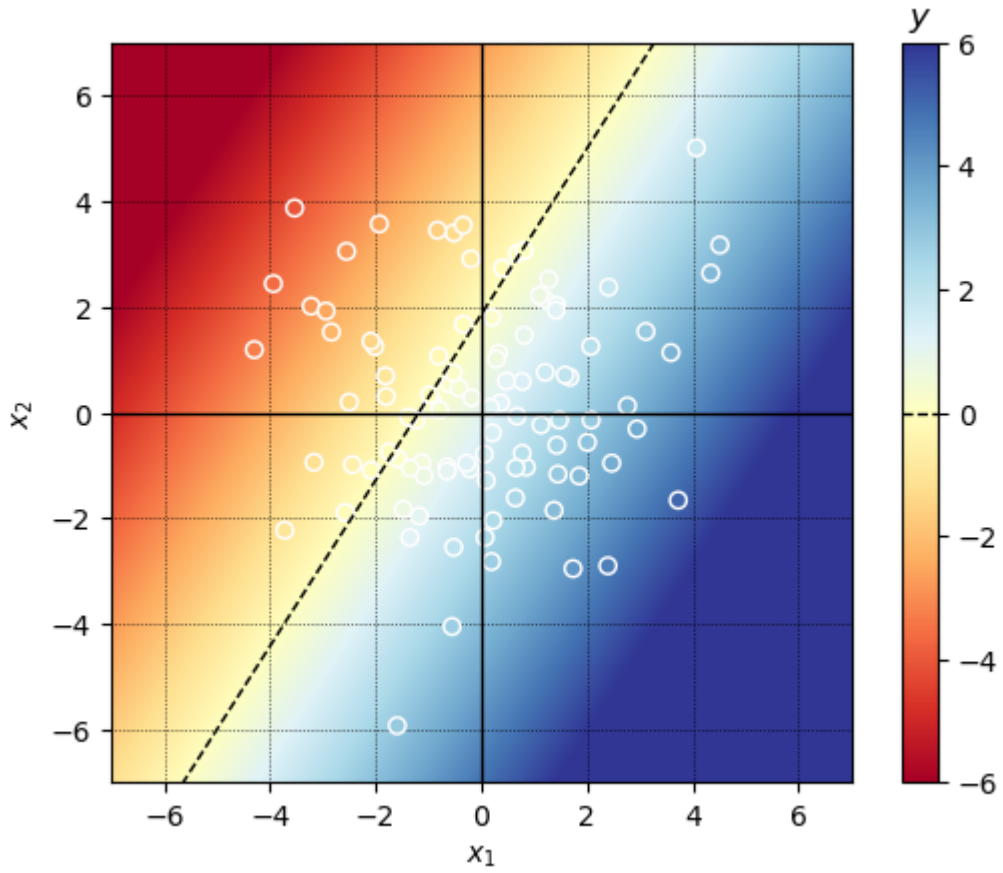
	x1	x2	y	ŷ
0	4.338429	2.637605	3.242041	3.242041
1	-0.653553	-1.027194	1.000647	1.000647
2	-0.937904	0.227610	0.086669	0.086669
3	-0.215532	-1.069881	1.393017	1.393017
4	3.723618	-1.659275	5.032026	5.032026

Gefeliciteerd!
Je hebt nu ook een algoritme geïmplementeerd dat lineaire regressie kan uitvoeren.

Ter vergelijking tonen we hieronder ook het lineaire regressiemodel uit de machine-learning module scikit-learn. Bekijk de syntax van deze functie. De bias en coefficienten zijn opnieuw toegankelijk via de `intercept_` en `coef_` variabelen.

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

In [18]:
```
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 = 0.9999999999999999
- weights = [ 0.84341877 -0.53725672]
```
