Thema 4.1 SE

Reader Machine Learning

Bart Barnard

september 2019

Inhoudsopgave

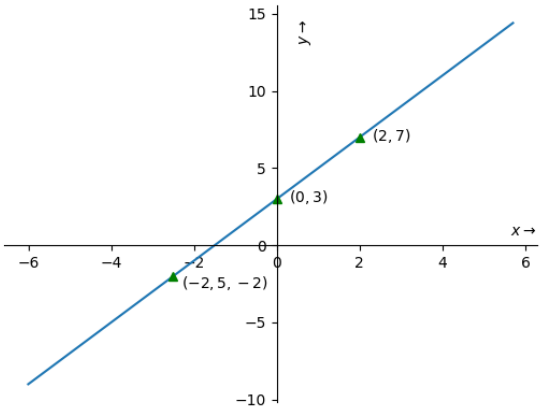
1 Functies en afgeleiden

1.1 Functies

Eén van de centrale concepten binnen de wiskunde, en zeker zeker ook binnen de *machine learning* is de *functie*. Formeel beschrijft een functie een relatie tussen twee verzamelingen X en Y, waarbij een element uit X gekoppeld wordt aan precies één element uit de verzameling Y. Eenvoudiger gezegd is een functie een verhouding tussen twee getallen, waarbij het ene getal (de zogenaamde *afhankelijke variabele*) wordt uitgedrukt in termen van het andere getal (de *onafhankelijke variabele*). Je kunt een functie opvatten als een machine die een geschikte invoer (geschikte getallen) omzet in een bepaalde uitvoer.

Een functie in de wiskunde doet op deze manier denken aan een functie (of *methode*) zoals we die kennen uit onze programmeerervaring. Zo wordt door de Java-methode `subString` een string omgezet in een betreffende substring, maar alleen als je deze aanroept met twee parameters van het type `int`. In methoden of functies waarin geen waarde wordt geretourneerd, is het return statement impliciet (elke methode of functie retourneert dus uiteindelijk een waarde).

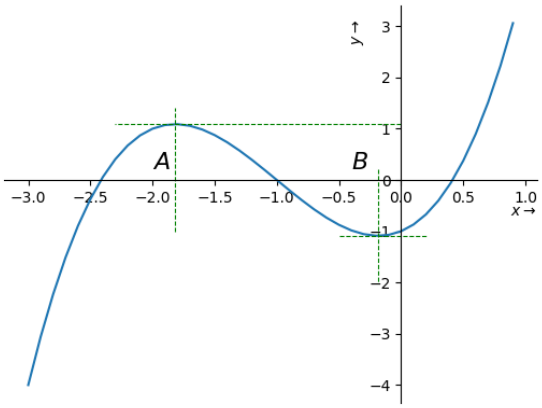
In de regel wordt de ongebonden variabele aangegeven met x en de gebonden variabele met y. Om aan te geven dat y van x afhankelijk is, schrijf je de *vergelijkingsvorm* $y = formule$, bijvoorbeeld $y = 2x + 3$. Een andere notatievorm is de *haakjesvorm*: $f(x) = 2x + 3$ – de functiewaarde van y is gelijk aan $2x + 3$. Behalve het geven van een formule kun je een functie natuurlijk ook in een assenstelsel tekenen. Hierbij zetten we de x op de horizontale lijn en de y op de verticale lijn. Elk punt in onze formule geven we dan weer als het *tupel* (x, y). Door al deze punten met elkaar te verbinden, verkrijgen we de grafiek van onze functie:



Figuur 1: Voorbeeld van een grafiek van een functie.

De verzameling van alle mogelijke waarden van de ongebonden variabele (de *input*) is het *domein* van de functie, dat we aangeven met D; de verzameling van de mogelijke waarden van de gebonden variabele (de *output*) is het *bereik* van die functie, wat wordt weergegeven met B. Gegeven een functie $f(x) = \sqrt{x-1}$, dan is het *domein* van die functie de verzameling van alle getallen groter dan of gelijk aan 1, terwijl het *bereik* van die functie gelijk is aan alle getallen groter dan of gelijk aan 0. Een dergelijke verzameling van getallen kun je omschrijven als een *interval*: als $a < b$ dan vormen alle getallen tussen a en b het *open interval* $\langle a, b\rangle$. Als de getallen a en b ook bij het interval horen, dan spreek je van een *gesloten interval*, wat je schrijft als $[a, b]$.

Een functie kan op een bepaald interval *stijgen*, *dalen*, of *gelijk blijven*. Een stijging of daling kan constant zijn, toenemen of afnemen. Het punt waarop een functie van een stijging overgaat naar een daling heet een *maximum*; het punt waar een functie overgaat van een daling naar een stijging heel een *minimum*. Zo heeft de functie in Figuur 2 een maximum in punt $(A, f(A))$ en een minumum in punt $(B, f(B))$. De functie vertoont een *afnemende stijging* tussen $[-\infty, A\rangle$ en een *toenemende stijging* tussen $\langle B, \infty]$; hij *daalt* tussen $[A, B]$. Verder is zowel het *domein* als het *bereik* van deze functie $[-\infty, \infty]$.

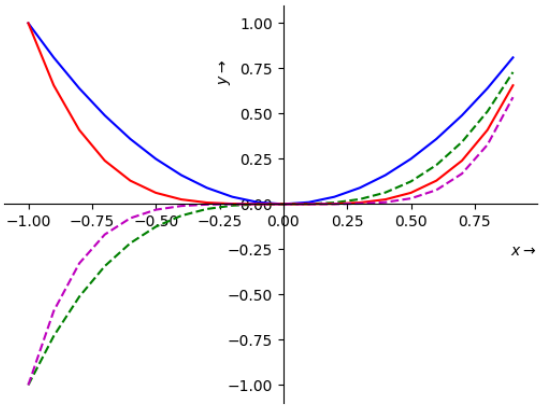


Figuur 2: Een functie met een minumum en een maximum.

1.2 Machtfuncties en polynomen

Een *machtsfunctie* is een functie waarvan de algemene vorm $f(x) = a \cdot x^n$ is, waarbij a (de *vermenigvuldigingsfactor* en n (de *exponent*) willekeurige positieve of negatieve getallen kunnen zijn. Wanneer de exponent geen geheel getal is, mag x niet negatief zijn. Wanneer de exponent een negatief getal is, mag x niet 0 zijn.

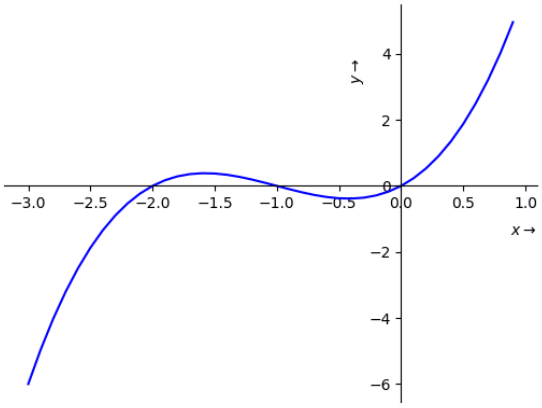
In Figuur 3 zijn de grafieken van x^2 (blauwe lijn), x^3 (groene stippellijn), x^4 (rode lijn) en x^5 (magenta stippellijn) getekend. Als je goed naar deze grafieken kijkt, valt een aantal zaken op. Alle vier de functies gaan door de punten $(0, 0)$ en door $(1, 1)$. Het bereik van de functies waarvan de exponent een *even getal* is, ligt boven de y-as ($f(x) \geq 0$ voor alle $x \in D$) en deze functies zijn *lijnsymmetrisch* in de y-as (dus $f(-x) = f(x)$ voor alle $x \in D$). Verder gaan deze functies allemaal door het punt $(-1, -1)$. De functies waarvan de exponent een *oneven getal* is, gaan allemaal door punt $(-1, -1)$ en zijn *puntsymmetrisch* in punt $(0, 0)$ (dus $f(-x) = -f(x)$ voor alle $x \in D$).



Figuur 3: Diverse machtsfuncties bij elkaar getekend.

Vaak zie je, zeker in het geval van machine learning, dat een functie bestaat uit een aantal termen met verschillende machten, zoals bijvoorbeeld de functie $f(x) = x^3 + 3x^2 + 2x$, die is weergegeven in Figuur 4. Zo'n soort functie noemen we een *polynoom* (van het Griekse polús (veel) en nomós (deel): een functie die uit veel delen bestaat). Je ziet hier dat $f(x) = 0$ voor $x = -2$, $x = -1$ en $x = 0$ (de *nulpunten*). Verder zitten er tussen de nulpunten een minimum en een maximum (hoewel die niet exact midden tussen de nulpunten in liggen). Wanneer we waarden van x nemen die heel groot of heel klein zijn, heeft de hoogste term in de functie-definitie de sterkste invloed op de waarden van $f(x)$ – de functie gaat nagenoeg gelijk lopen met de functie $g(x) = x^3$. Alleen dicht bij de 0 zijn er wat verschillen waar te nemen. Deze functie wordt een om die reden een polynoom van de *derde graad* genoemd: de hoogste exponent van de functie-definitie bepaalt de *graad* van de functie.

Net als bij enkelvoudige functies kunnen de termen in een polynoom voorzien worden van een vermenigvuldigingsfactor. Zo heeft de term x^2 in de polynoom in Figuur 4 een vermenigvuldigingsfactor van 3 en de term x een vermenigvuldigingsfactor van 2 (feitelijk is de vermenigvuldigingsfactor van de term x^3



Figuur 4: Een polynoom van de derde graad.

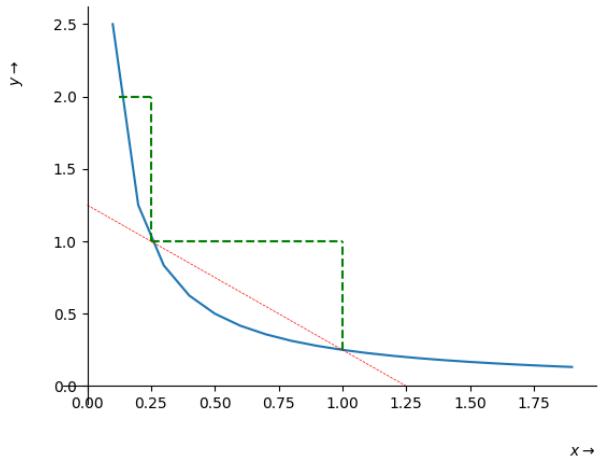
gelijk aan 1, dus dat laten we weg). Het is gebruikelijk om voor het beschrijven van deze factoren dezelfde letter te gebruiken, en die te voorzien van de term waar deze bij hoort als subscript. Zo wordt de algemene beschrijving van een polynoom gegeven door $f(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x^1 + a_0$: dit is dan een polynoom van de n-de graad.

1.3 Afgeleide functies

Vaak is het van belang om te weten of een functie op een interval $[a, b]$ stijgt of daalt – de zogenaamde *differentie*. Om dit te bepalen, kunnen we de waarde van $f(x)$ (of y) voor zowel a als b uitrekenen en van elkaar aftrekken. Als $f(a) < f(b)$, dan is de differentie van $f(x)$ op $[a, b]$ *negatief*: de functie vertoont een daling op het interval. Omgekeerd, als $f(a) > f(b)$, dan is de differentie van $f(x)$ op $[a, b]$ *positief*: de functie vertoont een stijging. De totale differentie geven we aan met de griekse hoofdletter delta: Δ.

In Figuur 5 is de functie $f(x) = \frac{1}{4x}$ op het interval $\langle 0, 2]$ weergegeven. Stel dat we de daling van deze functie op het interval $\left[\frac{1}{8}, \frac{1}{4}\right]$ willen weten, zoals aangegeven door de linker groene stippellijn. Om dit te bepalen, berekenen we eerst $f(x)$ voor het rechter extreem van het interval, en trekken daar $f(x)$ voor het linker extreem vanaf. De totale daling Δ wordt dan $f(\frac{1}{4}) - f(\frac{1}{8}) = \frac{1}{1} - \frac{1}{0.5} = 1 - 2 = -1$. Op eenzelfde manier kunnen we de totale differentie op het interval $\left[\frac{1}{4}, 1\right]$ bepalen: $\Delta = f(1) - f(\frac{1}{4}) = \frac{1}{4} - \frac{1}{1} = 0.25 - 1 = -0.75$.

Behalve het totaal is ook de *gemiddelde* stijging of daling van belang – het zogenaamde *differentiequotiënt*. Dit berekenen we door de totale stijging of daling op een interval te delen door de lengte van het interval. Het differentiequotiënt van $f(x)$ uit Figuur 5 op het interval $\left[\frac{1}{4}, 1\right]$ is dus $\frac{-0.75}{1-\frac{1}{4}} = -1$. Dit cijfer corres-



Figuur 5: De functie $f(x) = 1/4x$

pondeert met de *richtingscoëfficiënt* van de lijn die de beide extremen van het gegeven interval met elkaar verbindt – de rode stippellijn in Figuur 5. Algemeen geldt dat als $a < b$, het differentiequotiënt gelijk is aan

$$\frac{f(b) - f(a)}{b - a}.$$

Om de differentie van een functie op een specifiek punt $f(a)$ te bepalen, berekenen we de gemiddelde differentie van de functie op het interval $[a, a + h]$, waarbij we h steeds kleiner maken. Stel nu dat $f(x) = x^2$, dan kunnen we differentie op punt x als volgt bepalen:

$$\begin{aligned}
\Delta &= \frac{f(x+h) - f(x)}{(x+h) - x} && \text{waarden voor } x \text{ en } h \text{ invullen:} \\
&= \frac{(x+h)^2 - x^2}{(x+h) - x} && \text{kwadraten uitwerken en noemer versimpelen:} \\
&= \frac{(x^2 + 2xh + h^2) - x^2}{h} && \text{teller vereenvoudigen:} \\
&= \frac{2xh + h^2}{h} && \text{alles delen door d:} \\
&= 2x + h && \text{h nadert tot nul:} \\
&= 2x
\end{aligned}$$

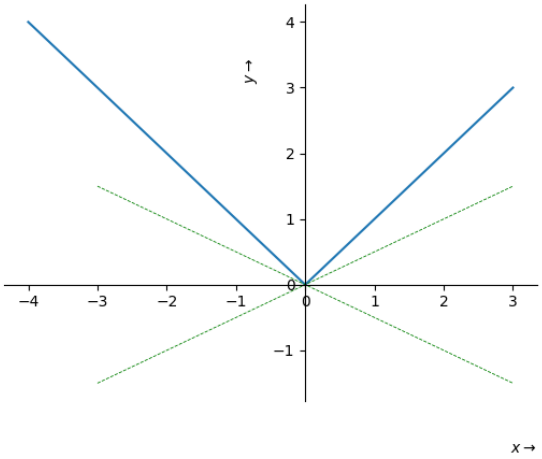
We zien dus dat de differentie van elk punt x op de functie $f(x) = x^2$ zelf ook weer een functie van x is, namelijk $2x$. Deze tweede functie heet de *afgeleide functie* van $f(x)$ en wordt genoteerd als $f'(x)$, of als een zogenaam *differentiequotiënt*:

$$f'(x) = \frac{df}{dx}$$

Het bepalen van een afgeleide functie wordt *differentiëren* genoemd; de grafiek van $f'(x)$ heet de *hellingsgrafiek* van $f(x)$. Differentiëren is een tak van de wiskunde die behoorlijk complex kan worden, maar voor nu volstaan de volgende algemene regels:

- De hellingsgrafiek van een constante functie $f(x) = x$ is de lijn $f'(x) = 0$: de lokale verandering van functiewaarden is overal 0.
- De hellingsgrafiek van een niet-horizontale lijn $f(x) = ax+b$ is de constante functie $f'(x) = a$: het lokale verandering van functiewaarden is overal hetzelfde.
- In het algemeen geldt dat als $f(x) = x^n$, dan is $f'(x) = nx^{n-1}$.

Wanneer we een functie $f(x)$ met domein D hebben, noemen we deze functie *differentieerbaar* wanneer we voor elke $x \in D$ een afgeleide waarde kunnen bepalen. Zo is de functie $f(x) = x^2$ differentieerbaar, want over het hele domein ($D = [-\infty, infty]$) kunnen we een afgeleide waarde bepalen, namelijk $f'(x) = 2x$. De functie $f(x) = |x|$ (zie Figuur 6) is daarentegen niet differentieerbaar: de afgeleide functie hiervan is $f'(x) = -1(x < 0)$ en $f'(x) = 1(x > 0)$, maar voor $x = 0$ is de afgeleide waarde ongedefinieerd: je kunt immers oneindig veel lijnen trekken die allemaal het punt $(0, 0)$ raken (in de figuur zijn twee voorbeelden getekend – de groene stippellijnen).



Figuur 6: De functie $f(x) = |x|$

2 Lineaire algebra

Lineaire algebra is een wiskundige discipline die zich bezighoudt met vectoren en matrices, vectorruimten en lineaire transformaties en die aan de basis staat van veel wetenschappelijk en industrieel programmeerwerk. Zo zijn bijvoorbeeld in *game development* lineaire transformaties onontkoombaar, wordt het dagelijks vliegverkeer mogelijk gemaakt door berekeningen op vectoren, en kun je televisieschermen en displays zien als matrices. Ook *machine learning* gaat voor een groot deel over lineaire algebra, en het is dus van belang dat je een goede basiskennis hebt van vectoren en matrices, en hoe je ermee werkt.

2.1 matrices

Op het meest basale niveau is een matrix een rechthoek van (vaak, maar zeker niet per se, gehele) getallen, zoals de matrix X hieronder:

$$X = \begin{bmatrix} 3 & 5 & 7 \\ 4 & 6 & 8 \end{bmatrix}.$$

De grootte of *dimensionaliteit* van een matrix wordt aangeduid door het aantal regels maal het aantal kolommen. Zo is X hierboven een 2×3-matrix (hij heeft twee regels en drie kolommen); algemeen hebben we het over een $m \times n$-matrix. In de regel wordt een hoofdletter gebruikt om de matrix zelf te definiëren, en wordt de corresponderende kleine letter gebruikt voor de individuele elementen. Die individuele elementen worden vervolgens door middel van subscripts i en j van elkaar onderscheiden, waarbij i correspondeert met de regel en j met de kolom. Dus $x_{12} = 5$, $x_{23} = 8$, enzovoort:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{bmatrix}.$$

Je kunt een matrix ook *transponeren*. Dat geeft je aan met een superscript T en houdt in dat je de regels en kolommen omdraait. Dus de getransponeerde matrix X is X^T en die is weer gelijk aan

$$X^T = \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

Als A een $m \times n$ matrix is, dan is A^T een $n \times m$ matrix.

2.2 vectoren

Wanneer we te maken hebben met een $m \times 1$ matrix, dan spreken we over een *vector*. Vectoren geven we aan met een kleine letter en het i-de element van een vector x geven we aan met x_i (er is hier dus maar één getal in het subscript,

omdat er immers maar één kolom is). Er bestaat een onderscheid tussen *kolomvectoren* en *rijvectoren*. Als er bijvoorbeeld sprake is van een column-vector

$$x = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$$

dan is de row-vector $x^T = \begin{bmatrix} 3 & 5 & 7 \end{bmatrix}$.

2.3 vierkante matrices

Vaak heb je te maken met een *vierkante matrix*, met een dimensionaliteit van $n \times n$. Hiervan zijn twee speciale gevallen die je moet kennen. Allereerst is er de *diagonaalmatrix*: een matrix waarvoor a_{ij} gelijk is aan 0 als $i \neq j$. Omdat alle elementen behalve de diagonaal nul zijn, wordt deze ook vaak geschreven met het woord *diag*:

$$diag(a_{11}, a_{22}, a_{33}, \ldots, a_{nn}) = \begin{bmatrix} a_{11} & 0 & 0 & \ldots & 0 \\ 0 & a_{22} & 0 & \ldots & 0 \\ 0 & 0 & a_{33} & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & a_{nn} \end{bmatrix}.$$

De tweede vierkante matrix die van belang is, is de *identiteitsmatrix* of de *eenheidsmatrix*, normaliter aangeduid met I_n. Dit is feitelijk een diagonaalmatrix met louter enen langs de diagonaal:

$$I_n = diag(1, 1, 1 \ldots, 1) = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}.$$

Vaak wordt een identiteitsmatrix zonder subscript aangegeven: in dat geval kan de grootte worden afgeleid van de context (of is deze niet relevant). De i-de kolom van een identiteitsmatrix is de eenheidsvector e_i.

Behalve deze twee vierkante matrices bestaan er ook nog verschillende vormen van *triangulaire matrices*, maar die zijn voor het onderwerp *machine learning* niet zo relevant. Een laatste speciale vorm van een vierkante matrix is de zogenaamde *symmetrische matrix*: een matrix A is symmetrisch wanneer geldt dat $A = A^T$; zo is de matrix

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 6 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

een symmetrische matrix. Voor een dergelijke matrix geldt dus dat $a_{ij} = a_{ji}$.

3 Rekenen met matrices

Wanneer we willen rekenen met matrices, kunnen we twee gevallen onderscheiden: we kunnen werken met verschillen *matrices*, of we kunnen werken met een matrix en een *scalar*.

3.1 Matrices met scalars

Je kunt een matrix vermenigvuldigen met of delen door een scalaire waarde. Ook kun je een scalaire waarde bij een matrix optellen of er juist van aftrekken. Stel dat λ een willekeurig getal is, en $A = (a_{ij})$ een willekeurige matrix, dan krijg je de *scalaire vermenigvuldiging* van A door elk element hiervan te vermenigvuldigen met λ:

$$3 \times \begin{bmatrix} 3 & 2 \\ 4 & 8 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} 3 \times 3 & 3 \times 2 \\ 3 \times 4 & 3 \times 8 \\ 3 \times 5 & 3 \times 1 \end{bmatrix} = \begin{bmatrix} 9 & 6 \\ 12 & 24 \\ 15 & 3 \end{bmatrix}$$

Delen werkt feitelijk op een vergelijkbare manier; natuurlijk moet je er hier ook rekening mee houden dat je niet door 0 kunt delen. Ook het optellen en aftrekken van scalaire waarden gaat op een deze manier, al komt dat in de praktijk zelden voor. Verder gelden de standaard-eigenschappen: als je een matrix vermenigvuldigt met 0 krijg je (uiteraard) een matrix waarvan alle elementen 0 zijn en een matrix vermenigvuldigd met 1 is de matrix zelf. Een speciaal geval is nog de *negatieve matrix* van $A = (a_{ij})$, die gedefinieerd is als $-1 \times A$, waarbij elk element ij uit $-A$ gelijk is aan $-a_{ij}$. Hieruit volgt

$$\begin{aligned} A + (-A) &= 0 \\ &= (-A) + A. \end{aligned}$$

3.2 Twee of meer matrices

Om twee of meer matrices te kunnen optellen of aftrekken, moeten ze *compatibel* zijn, dat wil in dit geval zeggen dat ze dezelfde dimensionaliteit hebben; het resultaat is dan ook weer een matrix van deze grootte. Bij deze operaties gaan we uit van de individuele elementen van de matrices. Wanneer bijvoorbeeld $A = (a_{ij})$ en $B = (b_{ij})$ twee $m \times n$ matrices zijn, dan is hun som $C = (c_{ij}) = A + B$ een $m \times n$ matrix die gedefinieerd is als $c_{ij} = a_{ij} + b_{ij}$. Mutatis mutandis geldt hetzelfde voor aftrekken, zoals in de onderstaande voorbeelden:

$$\begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 4 & 2 \\ 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 4 & 9 \\ 3 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 2 \\ 4 & 9 \\ 3 & 2 \end{bmatrix} - \begin{bmatrix} 4 & 2 \\ 2 & 4 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 5 \\ 3 & 1 \end{bmatrix}$$

De *vermenigvuldiging* van matrices is iets complexer. Wanneer we bijvoorbeeld A en B met elkaar willen vermenigvuldigen, kan dat alleen wanneer het aantal *kolommen* (in de voorbeelden hieronder aangegeven met k) van A gelijk is aan het aantal *rijen* (aangeduid met r) van B (ook deze matrices moeten, kortom, *compatibel* zijn, alleen betekent dat hier iets anders). Als $A = (a_{ij})$ een $m \times n$ matrix is en $B = (b_{jk})$ een $n \times p$ matrix, dan is hun product-matrix $C = AB = (c_{jk})$ een $m \times p$ matrix, waarvoor geldt:

$$c_{jk} = \sum_{j=1}^{n} a_{ij} b_{jk}$$

voor $i = 1, 2, \ldots, m$ en $k = 1, 2, \ldots, p$. We vermenigvuldigen, kortom, elk element ij uit A met elk element jk uit B en tellen die vermenigvuldigen bij elkaar op om element ik uit C te verkrijgen. In het voorbeeld hieronder vermenigvuldigen we een 2×3 matrix met een 3×2 matrix; het resultaat is dus een 2×2 matrix:

$$\begin{aligned}
\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 0 & 1 \\ 5 & 2 \end{bmatrix} &= \begin{bmatrix} r_1 k_1 & r_1 k_2 \\ r_2 k_1 & r_2 k_2 \end{bmatrix} \\
&= \begin{bmatrix} 1 \times 1 + 3 \times 0 + 2 \times 5 & 1 \times 3 + 3 \times 1 + 2 \times 2 \\ 4 \times 1 + 0 \times 0 + 1 \times 5 & 4 \times 3 + 0 \times 1 + 1 \times 2 \end{bmatrix} \\
&= \begin{bmatrix} 11 & 10 \\ 9 & 14 \end{bmatrix}
\end{aligned}$$

Matrices hebben *bijna* alle algebraïsche eigenschappen van gewone getallen. Het vermenigvuldigen van matrix met een (compatibele) identitiesmatrix levert de matrix zelf op:

$$I_m A = A I_m = A.$$

Het vermenigvuldigen van een matrix met nul levert een nul-matrix op:

$$A0 = 0.$$

Vermenigvuldigingen zijn associatief en distributief, maar niet commutatief.

$$\begin{gathered}
A(BC) = (AB)C, \\
A(B + C) = AB + AC, \\
AB \neq BA.
\end{gathered}$$

3.3 Matrices en vectoren

Omdat een vector feitelijk een $m \times 1$ matrix is (of een $1 \times m$ matrix als we het hebben over een rijvector), zijn de wiskundige operaties hierop bijna één op één over te nemen. Als bijvoorbeeld A een $m \times n$ matrix is en x een vector met lengte n, dan is Ax een vector van lengte m:

$$\begin{bmatrix} 1 & 3 & 2 \\ 4 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 5 \end{bmatrix} = \begin{bmatrix} r_1 k_1 \\ r1 k_2 \end{bmatrix}$$
$$= \begin{bmatrix} 1 \times 1 + 3 \times 0 + 2 \times 5 \\ 4 \times 1 + 0 \times 0 + 1 \times 5 \end{bmatrix}$$
$$= \begin{bmatrix} 11 \\ 9 \end{bmatrix}$$

Wanneer we werken met twee vectoren, krijgen we een interessante situatie. Stel dat x en y vectoren zijn met lengte m en dat we deze twee met elkaar willen vermenigvuldigen. Omdat we twee matrices met dimensionaliteit $m \times n$ en $p \times n$ niet zonder meer met elkaar kunnen vermenigvuldigen (ze zijn immers niet compatibel), moeten we één van die twee transponeren (zodat we bijvoorbeeld $m \times n$ en $n \times p$ krijgen). Dit geldt natuurlijk ook voor vectoren, maar als we de linker vector x transponeren, krijgen we feitelijk een $1 \times m$-dimensionale matrix. Als we die nu vermenigvuldigen met y is het resultaat een matrix van 1×1 – eigenlijk gewoon een *getal* dus. Dit getal is het *inproduct* van de vectoren x en y:

$$x^T y = \sum_{i=1}^{m} x_i y_i :$$
$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 \times 4 + 3 \times 5 + 3 \times 6 \end{bmatrix}$$
$$= 37.$$

De superscript T (in x^T) wordt soms ook weggelaten, als uit de context duidelijk is dat het om een inproduct gaat. Ook zie je wel het gebruik van de vermenigvuldigingspunt om dit nog explicieter aan te geven ($x \cdot y$).

Omgekeerd krijgen we als we de rechter vector y transponeren een $m \times m$ matrix die bekend staat als het *tensorproduct* (niet te verwarren met het *uitproduct*). Stelt dat Z de matrix is die verkregen wordt door xy^T, dan geldt $z_{ij} = x_i y_j$.

4 Opgaven en oefeningen

4.1 Matrices en vectoren

De opgaven hieronder zijn bedoeld als extra oefening voor het werken met matrices en vectoren. Ze vormen *geen* onderdeel van de module *machine learning* en worden dus ook niet getoetst. Mocht je echter moeite hebben met lineaire algebra, dan is het aan te bevelen deze opgaven door te lopen en eventuele problemen of vragen met je practicumdocent te bespreken.

1. Bereken de onderstaande opgaven:

   (a)
   $$\begin{bmatrix} 3 & 2 \\ 4 & 1 \\ 6 & 8 \end{bmatrix} \begin{bmatrix} 6 & 8 & 1 \\ 4 & 8 & 3 \end{bmatrix}$$

   (b)
   $$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

   (c)
   $$\begin{bmatrix} 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 1 & 2 \end{bmatrix}$$

   (d)
   $$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} w & x \\ y & z \end{bmatrix}$$

   (e)
   $$\begin{bmatrix} 3 \\ 1 \\ 4 \end{bmatrix} \begin{bmatrix} 5 & 9 & 2 \end{bmatrix}$$

   (f)
   $$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

2. Een wiskundige operator wordt *commutatief* genoemd wanneer de volgorde van de operatoren niet van invloed is op de uitkomst. Zo is $4+3 = 3+4$, dus optellen is commutatief. Maak gebruik van de eigenschappen matrices om te bewijzen dat het vermenigvuldigen van matrices *niet* commutatief is.

3. Matrix vermenigvuldigingen zijn associatief: $A(BC) = (AB)C$. Verifieer deze eigenschap door een drietal willekeurige 2×2 matrices met elkaar te vermenigvuldigen. Kun je aan de hand hiervan bedenken hoe je deze eigenschap zou kunnen bewijzen?

4. Verifieer dat $AI_m = A$ door een $m \times m$ matrix A te vermenigvuldigen met de identiteitsmatrix I_m. Kun je aan de hand hiervan bedenken hoe je deze eigenschap zou kunnen bewijzen?

5 Matrices en vectoren in NumPy

Om te kunnen werken met matrixes en vectoren gebruiken we in Python het package NumPy (`numpy.scipy.org`). Hoewel NumPy een datatype *matrix* heeft, maak je voor het werken met matrices in de regel gebruik van een array van arrays (de klasse *matrix* is depricated en zal worden uitgefaseerd). Een NumPy-array is een homogene multidimensionale array: een verzameling van elementen van hetzelfde type. Het datatype van een NumPy-array is `ndarray`, wat iets anders is dan de standaard Python-array (die is namelijk 1-dimensionaal en heeft minder functionaliteit).

5.1 Maken van matrices en vectoren

De dimensionaliteit van een ndarray wordt uitgedrukt in *assen* (*axis*): een 2 ×3 matrix wordt in een ndarray gerepresenteerd als een array van twee arrays (axis 1) met elk drie elementen (axis 2). Let op dat in NumPy arrays zero-based zijn, terwijl in de literatuur over lineaire algebra de verzamelingen meestal 1-based zijn. De dimensionaliteit van een ndarray kun je opvragen met de methode *shape()*. Je kunt een matrix of vector transponeren met behulp van de *property* `T`. De NumPy array API definieert vijf parameters bij het aanmaken van een ndarray, waarvan alleen de eerste verplicht is: hierin zit de data waarvan een ndarray gemaakt moet worden. Zie de voorbeelden hieronder:

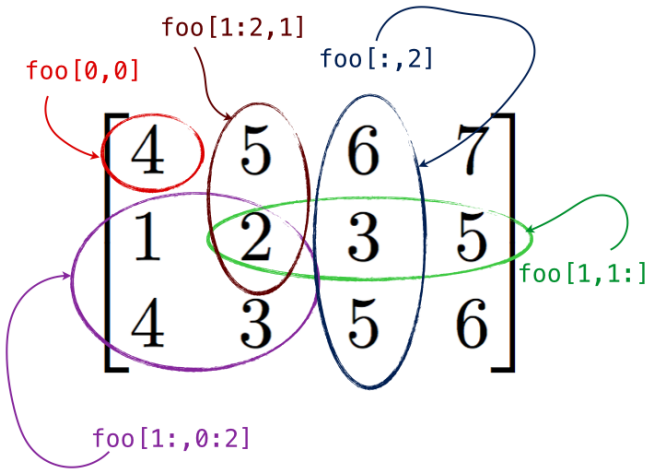
```
import numpy as np
v1 = np.array([2,3,5,7])
v2 = np.array([[2,3,5,7],[2,4,6,8]])
v1.shape()
> (4,)
v2.shape()
(2,4)
v2.T.shape
> (4, 2)
```

In deze voorbeelden bevat `v1` een 1-dimensionale array, terwijl `v2` een 2-dimensionale array bevat – de data is dus feitelijk een array met twee elementen, beide een array van vier elementen: `v2` kun je dus beschouwen als een 2 × 4 matrix. Let ook op de `shape` van `v1`: waar je misschien (1,4) zou verwachten, krijg je (4,). Dit komt omdat we een tupel hebben aangemaakt met maar één element, namelijk de array [2,3,5,7]; deze notatie-vorm (met een 'leeg' tweede element) is de standaard Python manier om dergelijke tupels weer te geven. Dit kan problemen opleveren wanneer je deze array als vector wilt gebruiken (bijvoorbeeld wilt transponeren of vermenigvuldigen met een andere vector of een matrix). Er zijn verschillende mogelijkheden om dit punt te adresseren:

```
np.array([[2,4,6,8]]).shape
> (1, 4)
np.array([1,3,5,7], ndmin=2).shape
```

```
> (1, 4)
```

Het benaderen van individuele elementen uit een vector of matrix gaat zoals je zou verwachten, met blokhaken en een *zero-based* index. Je gebruikt een dubbele punt om ranges aan te geven. Als je alleen een dubbele punt gebruikt, krijg je de hele rij of kolom te zien. Zie de voorbeelden in de afbeelding hieronder.



Je kunt met NumPy ook eenvoudig een identiteitsmatrix maken, of een matrix van een bepaalde grootte gevuld met nullen, enen, of willekeurige getallen (tussen 0 en 1, of met een bepaalde range van gehele getallen):

```
np.eye(3,4)             # identiteitsmatrix
np.zeros((2,3))         # nullen
np.ones((2,3))          # enen
np.random.rand(2,3)     # random tussen 0 en 1
np.random.randint(5, size=(2,3))
```

Om een kolom of een regel aan een reeds bestaande matrix toe te voegen, zijn er verschillende mogelijkheden. Zo kun je bijvoorbeeld gebruik maken van de methoden *hstack* of *vstack*, je kunt negatieve kolommen of rijen van een bepaalde waarde voorzien, of je maakt gebruik van `c_` of `r_`. De afweging tussen de alternatieven zit hem vooral in de flexibiliteit versus de snelheid versus het geheugenverbruik. In de voorbeelden hieronder worden deze drie methoden gebruikt om een nieuwe matrix `X2` te maken, die gelijk is aan `X` met een extra kolom van enen aan de linkerkant hiervan – vergelijkbare technieken kun je gebruiken om regels toe te voegen. Ga er van uit dat `X`, `m` en `n` telkens weer zijn vernieuwd.

```
X = np.random.randint(9, size=(8,6))
m,n = X.shape
>
```

```
#methode 1
foo = np.ones((m,1))
X2 = np.hstack((foo, X))
>
#methode 2
X2 = np.ones((m, n+1))
X2[:, 1:] = X
>
#methode 3
X2 = np.c_[np.ones(m), X]
```

5.2 Omvormen, kopiëren en sorteren

Stel je voor dat je een hele zooi data hebt die is opgeslagen als een $m \times n$ matrix – bijvoorbeeld verzameling van 2-dimensionale plaatjes of reistijdentabellen. Soms wil je dergelijke data in één grote matrix representeren, bijvoorbeeld om deze in één keer op te kunnen slaan of alle data in één keer aan een bepaalde methode door te geven. In dat geval is het handig om de afzonderlijke matrices om te vormen in een $mn \times 1$ kolomvector of een $1 \times nm$ rijvector en deze vectoren in een grotere matrix op te slaan. We gebruiken hier de methode *resize*: een methode die een array (of, dus, een matrix) een andere dimensionaliteit geeft. In het voorbeeld hieronder maken van we van twintig 8 $\times$5 (willekeurige) matrices een grote 20 $\times$40 matrix. Let op dat we de data eerst aan een lijst toevoegen en daarna de gehele lijst in één keer aan de constructor van de array meegeven.

```
foo = []
for _ in range(20):
    foo.append(np.random.randint(8, size=(8,5)))
newlist = []
for m in foo:
    newlist.append(np.resize(m, [1,40])[0])
X = np.array(newlist)
```

Als je twee instanties van dezelfde vector wilt hebben, moet je gebruik maken van de methode copy. Eenvoudig een tweede variabele gelijk stellen aan de eerste die naar de vector verwijst werkt niet, omdat je daarmee alleen de *pointers* gelijk maakt.

```
foo = np.array([ [2,3],[4,5] ])
bar = foo
bar[0][0] = 9
foo
> array([[9, 3],
>        [4, 5]])
#terwijl
foo = np.array([ [2,3],[4,5] ])
bar = np.copy(foo)
```

```
bar[0][0]=9
bar
> array([[9,  3],
>        [4,  5]])
foo
> array([[2,  3],
>        [4,  5]])
```

Om arrays te sorteren gebruik je de methode *sort*. Omdat een matrix ook een vector is, kun je deze methode ook gebruiken om matrices te sorteren. Het is dan wel van belang dat je aangeeft langs welke as (axis) je de matrix wilt sorteren: je kunt immers de *kolommen* of de *rijen* sorteren, zoals in het voorbeeld hieronder:

$$X = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 7 & 1 \end{bmatrix},$$
$$X_{hor} = \begin{bmatrix} 2 & 3 & 4 \\ 1 & 5 & 7 \end{bmatrix},$$
$$X_{ver} = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 7 & 4 \end{bmatrix}.$$

In NumPy bereik je dit effect als volgt:

```
X = np.array([ [2,3,4],[5,7,1] ])
np.sort(X, axis=-1) #default waarde voor axis
array([[2,  3,  4],
       [1,  5,  7]])
np.sort(X, axis=0)
array([[2,  3,  1],
       [5,  7,  4]])
```

5.3 Rekenen met matrices en vectoren

Je kunt gebruik maken van de gewone wiskundige operatoren in Python om berekeningen op vectoren uit te voeren. Standaard werken deze operatoren per element (*element wise*). Dat heeft tot gevolg dat als je een gewone operator gebruikt om twee matrices op te tellen of te vermenigvuldigen, deze dezelfde dimensionaliteit moeten hebben (elk element uit de linkermatrix moet gekoppeld kunnen worden uit een element uit de rechter).

```
foo = np.array([ [2,3,5], [1,3,7] ])
bar = np.array([ [4,5,6], [1,2,3] ])
foo + bar
> array([[ 6,  8, 11],
>        [ 2,  5, 10]])
foo - bar
> array([[-2, -2, -1],
>        [ 0,  1,  4]])
```

```
foo*2
> array([[ 4,  6, 10],
>        [ 2,  6, 14]])
foo**2
> array([[ 4,  9, 25],
>        [ 1,  9, 49]])
foo+2
> array([[4, 5, 7],
>        [3, 5, 9]])
```

Wanneer we twee matrices niet *element wise* met elkaar willen vermenigvuldigen, maar op de manier zoals dat bij de lineaire algebra is beschreven, moeten we gebruik maken van de NumPy-methode matmul (of, wat op hetzelfde neerkomt, `a @ b`) – let dan wel op de correcte dimensionaliteit van de matrices. Wanneer we twee vectoren met elkaar willen vermenigvuldigen, kunnen we gebruik maken van *dot*.

```
#twee matrices
X1 = np.array([ [2,3,5], [1,3,7] ])
X2 = np.array([ [4,5], [1,2], [5,6] ])
X1 @ X2
> array([[36, 46],
>        [42, 53]])
#maar
foo = np.random.randint(5, size=(2,3))
bar = np.random.randint(6, size=(4,5))
foo@bar
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
>   ValueError: shapes (2,3) and (4,5) not aligned: 3 (dim 1) != 4 (dim 0)
#twee vectoren
foo = np.array([2,3,4])
bar = np.array([6,7,8])
foo.dot(bar)
> 65
```

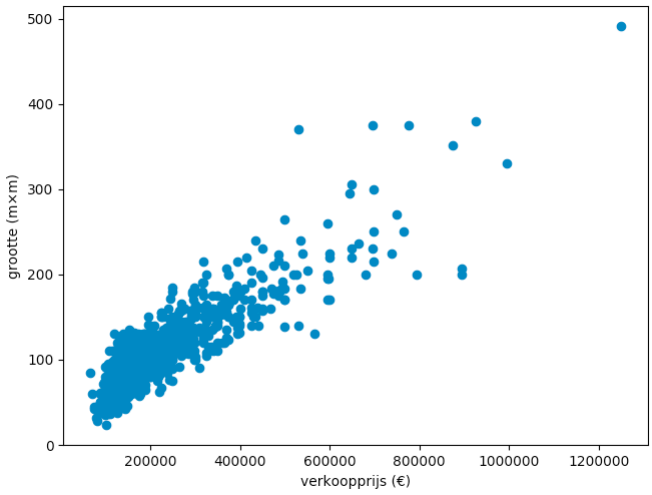
6 De kostenfunctie

Een groot deel van machine learning, zeker wanneer we het hebben over *gecontroleerd leren*, heeft betrekking op het voorspellen van de waarde van een variabele op basis van de waarde van een andere variabele. Stel je bijvoorbeeld voor dat we de verkoopwaarde van een huis willen voorspellen. Vanzelfsprekend zijn er legio aspecten die hierbij een rol spelen, maar een eerste eigenschap waar we naar zouden kunnen kijken is de grootte van het huis in kwestie. De vraag wordt dan of we op basis van de grootte van het huis iets kunnen zeggen over de waarde hiervan.

Overigens raken we hier direct aan een aardig aspect van *machine learning*, namelijk dat twee of meer variabelen prima een gelijk verloop kunnen vertonen, zonder dat hier een causaal verband tussen bestaat. Wanneer je aspecten van je data met elkaar in verband wilt brengen, moet je je er dus altijd van vergewissen dat hier een *logische* verhouding tussen bestaat. Je loopt anders het risico op zogenaamde *spurieuze verhoudingen*. Zie voor mooie voorbeelden hiervan `http://tylervigen.com/spurious-correlations`.

6.1 De som van de gekwadrateerde afwijking

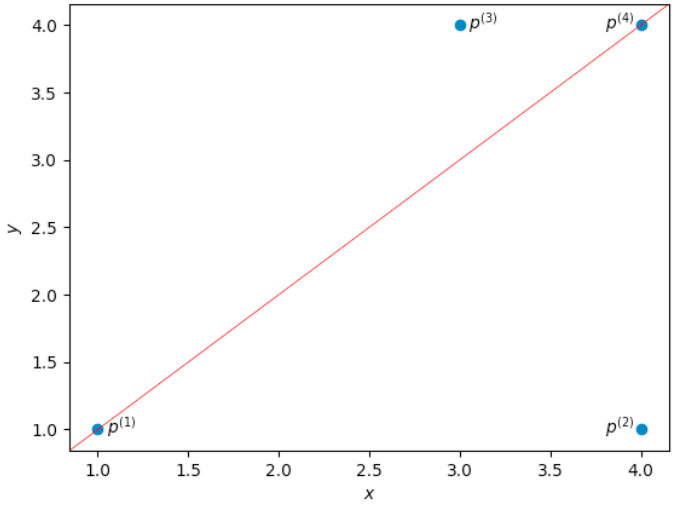
Wanneer we de verhouding tussen de grootte van een huis platten tegenover de waarde van datzelfde huis, krijgen we bijvoorbeeld een plaatje als in figuur 7:



Figuur 7: Scatterplot van de grootte van een huis ten opzichte van de prijs.

De scatterplot in figuur 7 toont een aantal *observaties*: elk punt (x, y) is een observatie van de prijs en de grootte van een huis. Zoals je ziet, is er wel een bepaalde correlatie tussen deze twee grootheden te identificeren: hoe groter het huis, hoe duurder. De vraag is nu hoe exact deze verhouding is.

Om de manier waarop je deze vraag kunt beantwoorden te verduidelijken, gebruiken we de wat eenvoudiger scatterplot in figuur 8. Hier zijn twee variabelen x en y tegenover elkaar gezet, met daarin vier observaties: $p^{(1)} = (1, 1)$, $p^{(2)} = (1, 4)$, $p^{(3)} = (3, 4)$ en $p^{(4)} = (4, 4)$. Verder stellen we als *hypothese* dat de waarde van y gelijk is aan de waarde van x, kortom dat $y = x$. Deze hypothese wordt in figuur 8 door de dunne rode lijn weergegeven.

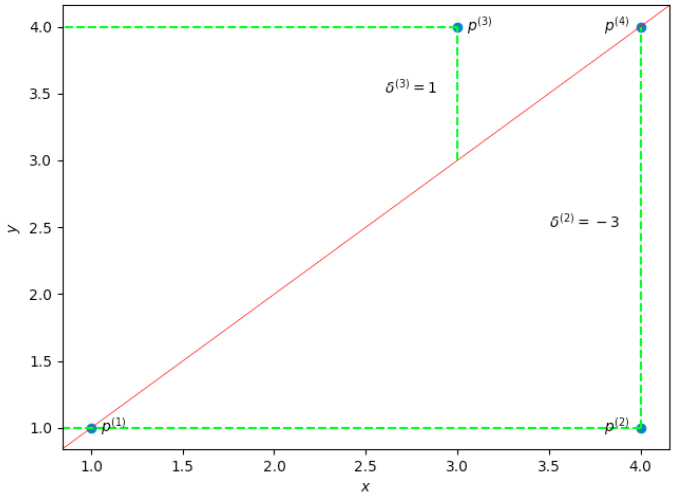


Figuur 8: Eenvoudige scatterplot met vier observaties.

Uitgaande van deze hypothese blijkt dat de punten $p^{(1)}$ en $p^{(4)}$ exact aan de voorspelde waarde voldoen, maar twee punten ook niet. Voor punt $p^{(2)}$ is de voorspelde waarde 4, maar de werkelijke waarde is 1; voor punt $p^{(3)}$ is de voorspelde waarde 3, maar is de werkelijke waarde 4. Het verschil tussen de voorspelde en de werkelijke waarde is dus -3 en 1 (zie figuur 9).

Wanneer we nu willen bepalen hoe goed onze hypothese is, moeten we al deze verschillen tussen de voorspelde en werkelijke waarde met elkaar optellen. Een probleem hierbij is echter dat deze verschillen zowel positief als negatief kunnen zijn. Hierdoor zou de som van al deze verschillen lager kunnen uitpakken dan werkelijk het geval is: zo is in het voorbeeld in figuur 9 de som van alle verschillen gelijk aan $1 + -3 = 2$, terwijl we intuïtief aanvoelen dat deze som eerder $1 + 3 = 4$ zou moeten zijn.

Om dit probleem te adresseren tellen we de *kwadraten* van al deze verschillen bij elkaar op, zodat ook negatieve verschillen een positieve bijdrage leveren. Om te bepalen hoe goed de hypothese is, delen we tenslotte deze som door het aantal datapunten, zodat we een *gemiddelde afwijking* krijgen. Deze afwijking vormt de *kost* van onze hypothese: feitelijk beschrijft dit wat het kost om met een bepaalde hypothese een set van datapunten te beschrijven. Hoe lager deze kosten hoe beter de hypothese is.



Figuur 9: Het verschil tussen voorspelde en actuele waarden.

$$KOST = \frac{0^2 + (-3)^2 + 1^2 + 0^2}{4}$$
$$= \frac{0 + 9 + 1 + 0}{4} = 2,5$$

6.2 Gegevensnotatie

Om hier goed over te kunnen redeneren en mee te kunnen rekenen is een conventie nodig met betrekking tot de notatie van de verschillende elementen.

In het voorbeeld hierboven is steeds gesproken over slechts één eigenschap waar naar gekeken wordt om de prijs van een huis te bepalen: de grootte van het woonoppervlak. In werkelijkheid zijn meestal meer eigenschappen van data die je in je berekeningen mee wilt nemen. Deze eigenschappen (Engels: *features*) worden weergegeven door middel van een subscript j. De individuele observaties (*samples*) x uit de volledige trainingsdata X worden beschreven met een superscript i tussen haakjes. Dus $x_4^{(2)}$ heeft betrekking op de waarde van de *vierde* eigenschap van het *tweede* observatie. Stel je voor dat de trainingsdata bestaat uit vier eigenschappen, dan geldt dus:

$$x^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix} \in X.$$

Het totaal aantal observaties in de trainingsdata wordt weergegeven met m en de verwachte uitkomst met y. Voor elk sample $x^{(i)} \in X$ (waarvoor geldt $1 \leq i \leq m$) is er een corresponderende waarde $y^{(i)}$.

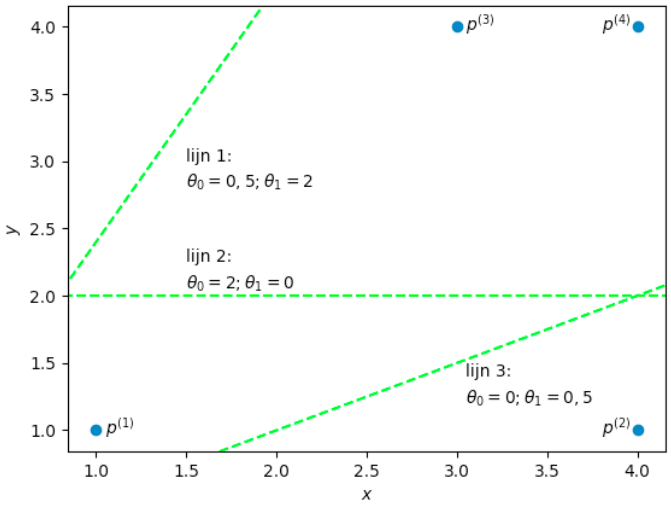
De hypothese h die we formuleren geeft als het ware een gewicht aan alle eigenschappen (*features*) van de trainingsset. Deze gewichten worden weergeven

met de Griekse kleine letter theta, θ, met een subscript om aan te geven bij welke eigenschap dat specifieke gewicht hoort. Het aantal eigenschappen van een dataset wordt aangegeven met n, dus θ is een $n \times 1$ kolomvector. Aan deze vector wordt meestal nog een initiële waarde θ_0 toegevoegd (zoals de algemene formule van een lijn: $y = b + ax$).

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_n \end{bmatrix}$$

In het huizen-voorbeeld is er maar één eigenschap waar naar gekeken wordt, dus de hypothese is in dit geval $h(x) = \theta_0 + \theta_1 x$. In de grafiek in figuur 10 is een aantal hypothesen met verschillende waarden voor θ getekend. Elk van deze lijnen correspondeert met een bepaalde hypothese over y gegeven x en een bepaalde waarde van θ. Meer in het algemeen zijn deze hypothesen dus functies van x die afhankelijk zijn van specifieke waarden van θ_0 en θ_1. Zo is lijn 1 in de grafiek gegeven door $h_\theta(x) = 0,5 + 2x$, lijn 2 door $h_\theta(x) = 2$ en lijn 3 door $h_\theta(x) = 0,5x$. De hypothese, kortom, van de waarde van y is gelijk aan elke waarde van θ vermenigvuldigd met de corresponderende waarde van de eigenschap van x, oftewel

$$h_\theta(x) = \theta^T x.$$



Figuur 10: Een aantal mogelijke hypothese.

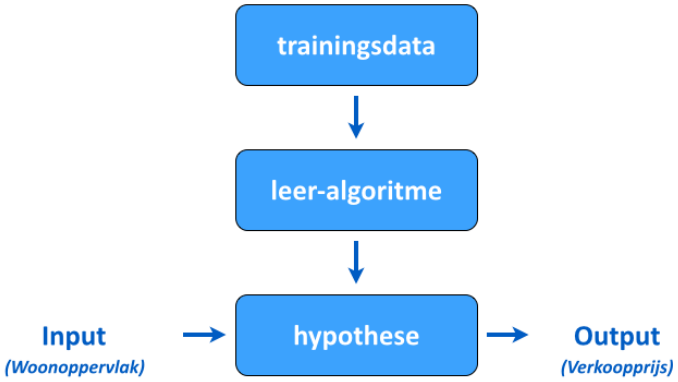
De vector θ beschrijft een hypothese van de verhouding tussen de gegeven en de gevraagde uitkomst. Voor elke observatie $x^{(i)}$ in de trainingsdata X wordt aan

de hand van θ de waarde van $h_\theta(x^{(i)})$ bepaald. Deze waarde wordt vergeleken met de *actuele* waarde $y^{(i)}$). Het verschil tussen deze twee wordt gekwadrateerd en al deze verschillen worden bij elkaar opgeteld en gedeeld door het aantal samples om de gemiddelde kosten te krijgen die de huidige hypothese met zich meebrengt. Deze kosten worden weergegeven met J, wat dus een een functie is van θ. De complete formule van de kostenfunctie is dus:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2.$$

6.3 Lineaire regressie

Het doel van lineaire regressie is de totale kosten $J(\theta)$ te minimaliseren: hoe lager deze waarde, hoe beter immers de hypothese overeenkomt met de werkelijke data. De algemene werking van *lineaire regressie* staat in figuur 11 schematisch afgebeeld. Op basis van een dataset gebruiken we een leer-algoritme om een hypothese te formuleren over de verhouding tussen de gegeven en de gevraagde data. Op basis van deze hypothese kunnen we op basis van *nieuwe* input (het woonoppervlak van een nieuwe aangeboden huis in Groningen) een voorstelling doen van de verwachte *output* (de verwachte verkoopwaarde). Hoe beter de hypothese de gegeven data beschrijft, hoe beter de voorspellende waarde hiervan (hoewel hier ook wel een optimum aan zit; daar komen we later over te spreken wanneer we het gaan hebben over *overfitting*).

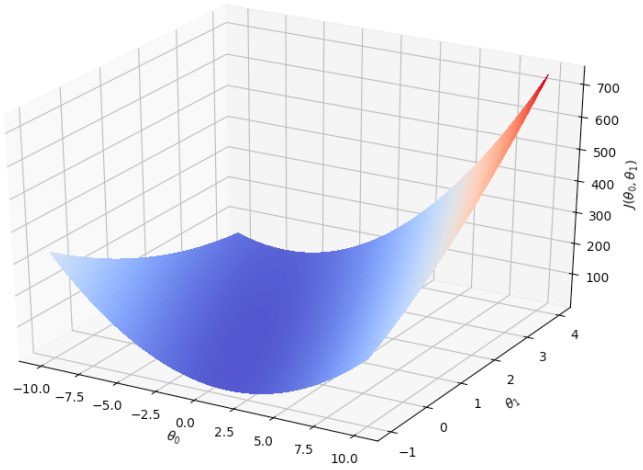


Figuur 11: Lineaire regressie.

De vraag die dus voorligt is dus welke waarden de vector θ moet bevatten om $J(\theta)$ zo klein mogelijk te maken. Wanneer we ons voor het gemak even houden bij het huizenvoorbeeld van hierboven, wordt de vraag welke waarden θ_1 en θ_2 moeten hebben om de lijn de definiëren die het beste voorspelling van de prijs doet op basis van de grootte van het huis. Het algoritme dat hiervoor bestaat is betrekkelijk eenvoudig: kies een waarde voor θ_1 en θ_2, bereken $J(\theta)$

voor deze waarden en verander θ_1 en θ_2 zodat $J(\theta)$ kleiner wordt. Het probleem is natuurlijk hoe we deze veranderingen uit moeten rekenen.

In het huizenvoorbeeld wordt de hypothese van de prijs van een huis gedefinieerd door $h_\theta(x) = \theta_1 + \theta_2 x$, waarbij x de grootte van het huis is kwestie is. Op basis van de kostenfunctie, kunnen we een plot tekenen van $J(\theta)$ voor een grote *range* van θ. Omdat we twee variabelen hebben, wordt dit een driedimensionaal figuur zoals in figuur 12.



Figuur 12: Contour plot van drie dimensies.

Voor elke waarde van θ kunnen we dus de hoek van de plot op dat punt bepalen, en die hoek kunnen we gebruiken voor onze update. We weten dat de hoek van een functie op een bepaald punt gegeven wordt door de *afgeleide* van die functie te nemen. Dat geldt ook voor een functie die afhankelijk is van twee (of meer) variabelen, alleen moeten we in dat geval werken met de *partiële* afgeleide. Wanneer we de snelheid van het algoritme weergeven als α, dan wordt het algoritme dat we moeten uitwerken als volgt:

Herhaal tot een minimum is bereikt
(voor j=0 en j=1){

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

}

Van belang is dat je in dit stappenplan alle waarden van j tegelijkertijd update. Dus je rekent eerst alle waarden uit en doet dan in één keer de update – anders krijg je een doorrekening van halve θ's. Omdat we meestal met meer dan één eigenschap (*feature*) van de data te maken hebben, is het gebruikelijker (en makkelijker) om elke data-vector uit te breiden met een extra 1 aan het

begin, zodat de dimensionaliteit van θ en van de data aan elkaar gelijk wordt – θ_0 wordt dan eenvoudig vermenigvuldigd met die extra 1. Aldus uitgebreid en veralgemeniseerd, wordt het algoritme voor lineaire regressie

Herhaal tot een minimum is bereikt
(gelijktijdige update voor alle θ_j, $j = 0 \ldots n$){

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

$$:= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

}.

7 Classificatie en logistische regressie

In het vorige hoofdstuk zijn we ingegaan op de vraag hoe we op basis van een bepaald aantal gegevens een voorspelling kunnen doen van de waarde van een ander gegeven – hoe, bijvoorbeeld, de waarde van een huis samenhangt met de grootte, of hoe de omzet van een vervoerder samenhangt met de stad waarin hij opereert. Een andere belangrijke tak van sport binnen machine learning betreft de zogenaamde *classificatie-problematiek*, waarbij we observaties onderverdelen in een set van categorieën.

Een mooi voorbeeld van een dergelijk vraagstuk wordt gevormd door e-mails. Stel je voor dat we een aantal categorieën hebben waarin we onze binnenkomende e-mails onderverdelen: ongewenste reclame natuurlijk, maar ook werk-gerelateerd, privé of mails van onze vrienden. Kunnen we nu een systeem trainen dat op zoek gaat naar verschillende eigenschappen van die binnenkomende mail en op basis hiervan deze mail automatisch in de juiste categorie onderverdeelt?

Meer algemeen kunnen we classificatie als volgt omschrijven: gegeven een set X van m observaties, kunnen we een systeem maken van voor een nieuwe observatie x een hypothese $h(x)$ formuleert of deze observatie *wel* ($y = 1$) of *niet* ($y = 0$) tot een bepaalde categorie behoort? Een systeem dat hierover uitspraken doet wordt een *classificeerder*(of in het Engels *classifier*) genoemd.

7.1 De sigmoïdefunctie

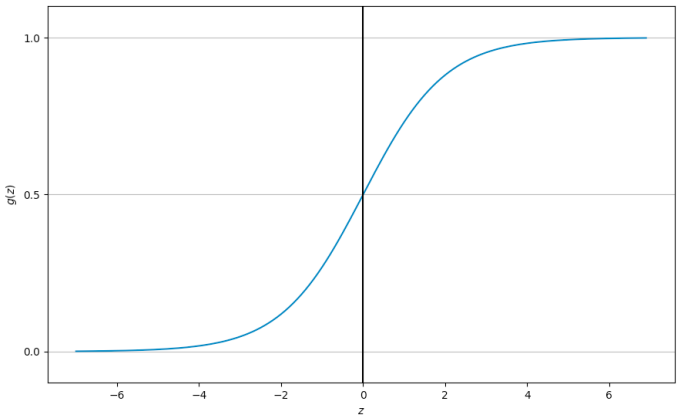
In tegenstelling tot de eerdere problemen, wordt bij classificatie de mogelijke output dus beperkt tot nul of één. Hoewel er natuurlijk een grijs gebied is, kun je van de meeste e-mails wel direct zeggen of het evident spam is of niet. Om deze situatie te kunnen modelleren, moeten we dus een functie hebben die 0 heeft als laagste waarde, 1 als hoogste waarde en een relatief snelle overgang tussen deze twee. De functie die aan deze eigenschappen voldoet, is de *sigmoïdefunctie*:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Zoals je in figuur 13 ziet is in deze functie $y \approx 1$ voor relatief hoge waarden van x, en $y \approx 0$ voor relatief lage waarden van x. De functie gaat door het punt $(1, \frac{1}{2})$, want $e^0 = 1$.

We gebruiken nu deze sigmoïdefunctie om de kans uit te rekenen of data tot een bepaalde categorie behoort of niet. Wanneer we bijvoorbeeld een lijst hebben van woorden die vaak in spam e-mails voorkomen, zouden we eenvoudig kunnen tellen hoe vaak deze woorden in een specifiek bericht voorkomen en op basis daarvan kunnen aangeven hoe *waarschijnlijk* het is dat dit bericht een spam-bericht is. Stel nu dat we een bericht hebben waarin drie van dergelijke woorden voorkomen. Een berekening zou er dan als volgt uit kunnen zien:

$$x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 1 \\ aantalWoorden \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}.$$



Figuur 13: De sigmoïdefunctie

Als we vervolgens stellen dat we hebben gevonden dat $\theta_0 = -0,1$ en $\theta_1 = -0,2$, dan geldt

$$\begin{aligned} z = \theta^T x &= \begin{bmatrix} -0,1 & -0,2 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} \\ &= -0,1 + -0,6 = -0,7. \end{aligned}$$

De waarschijnlijkheid dat dit bericht een spam-bericht is, wordt dan gegeven door

$$\begin{aligned} h_\theta(x) = g(\theta^T x) &= \frac{1}{1+e^{-z}} \\ &= \frac{1}{1+e^{0,7}} \approx \frac{1}{1+2,01} \\ &\approx \frac{1}{3,01} \approx 0,33. \end{aligned}$$

Met deze gegevens is er dus een kans van zo'n drieëndertig procent dat het bericht in kwestie een spam-bericht is. De vraag is natuurlijk, net als bij de lineaire regressie, hoe we tot de waardes van θ komen.

7.2 De kostenfunctie voor logistische regressie

Net als bij lineaire regressie hebben we ook bij logistische regressie te maken met een trainingsset van m observaties: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$, waarbij $y \in \{0, 1\}$ en $x \in \{x_0, x_1, \ldots, x_n\}$ (elke x heeft n eigenschappen en $x_0 = 1$). Voor elke $x^{(i)}$ is er een hypothese die aangeeft of deze observatie wel of niet tot de specifieke categorie behoort. Deze hypothese wordt gegeven door de sigmoïdefunctie:

$$h_\theta(x^{(i)}) = \frac{1}{1+e^{\theta^T x^{(i)}}}$$

Wanneer we echter deze functie zouden gebruiken om de som van de gekwadrateerde afwijkingen te bepalen, krijgen we een zeer complexe berekening waarvan niet te garanderen is dat deze ook daadwerkelijk te minimaliseren is. In het geval van de sigmoïdefunctie maken we daarom gebruik van een andere kostenfunctie, die is onder te verdelen in twee situaties: één waarbij $y = 0$ en één waarbij $y = 1$. Omdat we weten dat één van deze twee gegarandeerd het geval is, kunnen we deze twee situaties echter direct tot één formule reduceren:

$$\begin{aligned} Kost(h_\theta(x), y) &= \begin{cases} -log(h_\theta(x)) & \text{als } y = 1 \\ -log(1 - h_\theta(x)) & \text{als } y = 0 \end{cases} \\ &= -ylog(h_\theta(x)) - (1 - y)log(1 - h_\theta(x)). \end{aligned}$$

Wanneer we deze formule toepassen, zien we dat in het geval dat $h_\theta(x) \to 1$ en $y = 1$ (wanneer we, kortom, correct voorspellen dat de observatie x inderdaad tot deze specifieke categorie behoort) de *kost* voor deze voorspelling nadert naar 0 ($log(1) = 0$). Omgekeerd, wanneer $h_\theta(x) \to 0$ (terwijl $y = 1$), nadert deze kost oneindig ($log(0) = -\infty$). Mutatis mutandis geldt hetzelfde voor de situatie wanneer $y = 0$. Dit klopt natuurlijk ook met onze intuïtie: wanneer we een correcte voorspelling doen, willen we dat de kost van de hypothese die dat weergeeft minimaal is; als we een incorrecte voorspelling doen, is deze kost daarentegen erg hoog. Wanneer we nu de kosten voor alle observaties bij elkaar optellen en delen door het aantal observaties, dan krijgen we dus weer de gemiddelde kosten voor deze specifieke set van θ:

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

Net als bij lineaire regressie is het nu het doel om deze kosten tot een minimum te reduceren. Om dit te bereiken moeten we weer de afgeleide van de formule hierboven hebben. Hoewel hij er wellicht indrukwekkend uitziet, is de afgeleide opvallend eenvoudig (en herkenbaar):

Herhaal tot een minimum is bereikt
(gelijktijdige update voor alle θ_j, $j = 0 \dots n$){

$$\begin{aligned} \theta_j &:= \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &:= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \end{aligned}$$

}.

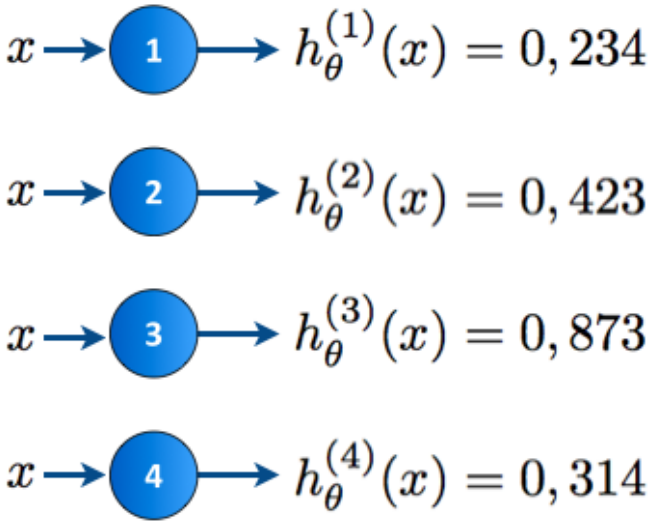
Kortom, de update-regel is voor logistische regressie hetzelfde als voor lineaire regressie. Het enige – belangrijke – verschil is dat we in het dit geval de sigmoïde-functie gebruiken om $h_\theta(x)$ uit te rekenen, terwijl we bij lineaire regressie een eenvoudige lijn gebruiken.

7.3 Classificatie met meerdere klassen

De methoden die we hierboven hebben beschreven, kunnen we gebruiken voor zogenaamde *binaire classificatie*: een observatie behoort tot een bepaalde klasse toe of niet. Hoe kunnen we nu deze technieken gebruiken om data onder te verdelen in meerdere klassen, bijvoorbeeld om e-mails te sorteren op spam, werkgerelateerd, privé of van vrienden?

Stel je voor dat we een situatie hebben waarin we data willen onderverdelen in vier categorieën. Onze trainingsdata bevat in dat geval $y \in 0, 1, 2, 3$: voor elke observatie is hierin aangegeven tot welke categorie deze behoort. Op de manier die hierboven beschreven is, kunnen we voor elke categorie een separate classificeerder trainen, die een kans geeft of een observatie tot deze specifieke klasse behoort of niet. Wanneer we dan een nieuwe observatie hebben, kiezen we die classificeerder die de hoogste waarschijnlijkheid aangeeft.

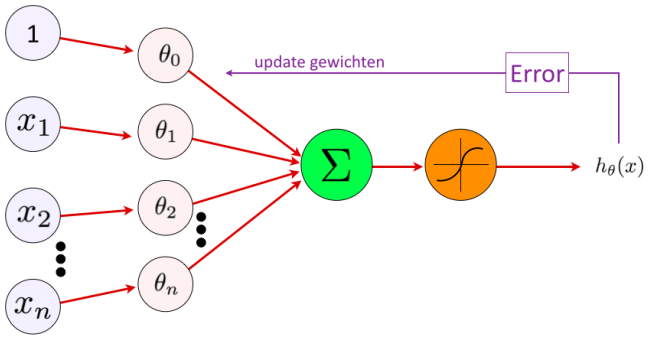
Wanneer we te maken hebben met m klassen, dan trainen we dus m classificeerders die voor elke nieuwe observatie $(x^{(i)}, y^{(i)})$ aangeven hoe groot de kans is dat $y = m$. Zie ook figuur 14, waarin de kans het grootst is dat de observatie behoort tot de derde klasse.



Figuur 14: Classificatie over meerdere categorieën.

8 Classificatie met neurale netwerken

Op een abstract niveau kunnen we zowel de lineaire als de logistische regressie als volgt omschrijven. We combineren een observatie x met een set van gewogen waarden θ om een hypothetische output $h_\theta(x)$ uit te rekenen. Deze hypothese checken we met de werkelijke waarde y en het verschil dat we hierbij vinden gebruiken we om de gewichten van θ aan te passen. Deze cyclus herhalen we totdat we vinden dat het verschil tussen onze hypothese en de daadwerkelijke waarde nagenoeg nul (of een binnen ons domein acceptabel minimum) is. Figuur 15 geeft dit proces schematisch weer. Dit plaatje, dat al in de jaren vijftig van de vorige eeuw is door F. Rosenblatt is ontwikkeld, staat bekend onder de naam *perceptron* en vormt de basis voor *artificiële neurale netwerken*.



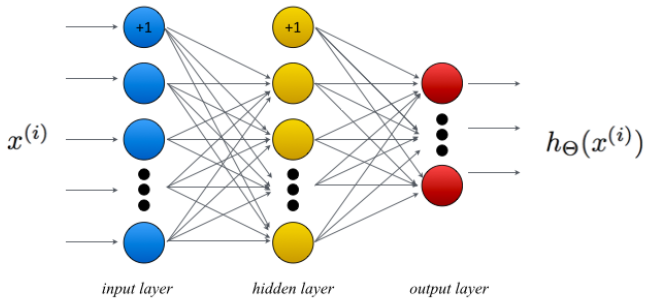
Figuur 15: De perceptron.

De cirkel in figuur 15 waarin Σ staat, kun je zien als een neuron die een getal tussen 0 en 1 teruggeeft, afhankelijk van de som van de gewogen input die deze cirkel binnenkrijgt van $x_1, x_2 \ldots, x_n$. Een neuraal netwerk bestaat uit een aantal van dergelijke neuronen (*nodes*) die allemaal op basis van de gewogen som van de input een 1 of een 0 als output genereren. De perceptron in deze afbeelding zou je kunnen zien als een neuraal netwerk met n *input nodes* en 1 *output node*.

8.1 Forward en backpropagation

Wanneer we het hebben over neurale netwerken, gaat het meestal over netwerken die één of meer zogenaamde *verborgen lagen*(*hidden layers*) hebben. Een dergelijke laag heet 'verborgen', omdat hij tussen de input- en de output-laag in zit. Elke node in de verborgen laag fungeert als een perceptron: hij is verbonden met alle nodes uit de input-laag en geeft op basis van de gewogen input een sigmoïde waarde tussen 0 en 1 door aan de volgende laag - zie figuur 16. Wanneer een dergelijk netwerk meer dan één verborgen laag bevat, spreken we over *deep learning*; het bepalen van het optimale aantal verborgen lagen en de hoeveelheid

nodes per laag is onderdeel van het zogenaamde *hyperparameter tuning* – een onderwerp waar we hier verder niet op in zullen gaan.



Figuur 16: De algemene structuur van een neuraal netwerk.

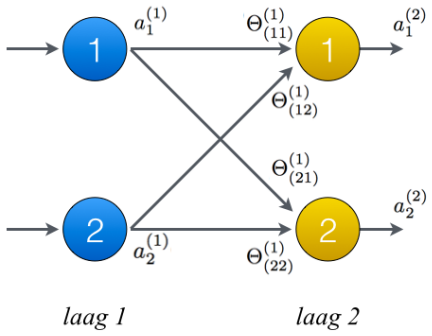
De sigmoïde output van node i in laag l wordt aangeduid met $a_i^{(l)}$ (a van *activation*). Omdat elke node in laag l verbonden is met elke node in laag $l+1$, vormen de gewichten tussen deze nodes een matrix; deze matrix wordt aangeduid met de Griekse hoofdletter theta: Θ. Het superscript van deze Θ geeft aan tussen welke lagen deze specifieke matrix geldt: de gewichten tussen de nodes in laag l en laag $l+1$ zijn opgeslagen in $\Theta^{(l)}$. In het subscript wordt aangegeven tussen welke nodes in die specifieke laag dat gewicht geldt: het gewicht tussen node i in laag l en node j in laag $l+1$ is dus weergegeven in $\Theta_{ji}^{(l)}$.

De activatie van een specifieke node wordt bepaald aan de hand van de sigmoïdefunctie van de som van de gewogen input te berekenen. Zie figuur 17, waarin een inputlaag van 2 nodes verbonden is met een verborgen laag van eveneens 2 nodes. Wanneer we de sigmoïdefunctie van z definiëren als $g(z)$, dan is de activatie (of output) van de tweede node in de tweede laag

$$a_2^{(2)} = g(\Theta_{21}^{(1)} \cdot a_1^{(1)} + \Theta_{22}^{(1)} \cdot a_2^{(1)}).$$

De classificatie van de input-data wordt dan bepaald door welke node in de output-laag het hoogste resultaat heeft voor deze specifieke input. Stel je bijvoorbeeld voor dat je een neuraal netwerk wilt trainen om e-mails te kunnen classificeren als spam, werk-gerelateerd, privé of familieperikelen. In dat geval zou de output-laag van het netwerk bestaan uit vier nodes, die elk één van deze categorieën representeert. Wanneer nu blijkt dat de eerste node de hoogste waarde van deze vier heeft wanneer we het netwerk een bepaalde e-mail als input geven, classificeren we deze email als spam. Om een dergelijk netwerk te kunnen trainen, hebben we dus trainingsdata nodig waarbij de input bestaat uit een zooi e-mails (platte tekst bijvoorbeeld) en de output bestaat uit een 4×1 kolomvector die voor al deze e-mails aangeeft tot welke categorie elke e-mail behoort.

De algemene werking van een neuraal netwerk kan dan in drie stappen worden samengevat:



Figuur 17: De berekening van de activatie van een node.

1. We gebruiken de trainingsdata als input voor het neurale netwerk. Aan de hand van deze data $x^{(i)}$ wordt op de manier die hierboven beschreven is de uiteindelijke output van het netwerk berekend. Deze stap staat bekend onder de term *forward propagation*.

2. Deze output wordt vergeleken met de verwachte waarde bij die specifieke input (de waarde van $y^{(i)}$) waardoor de we fout $\delta^{(i)}$ kunnen berekenen.

3. Nu draaien we de richting van het netwerk als het ware om; we sturen de δ terug het netwerk in (van de de output-laag naar de eerste laag) en passen de individuele gewichten tussen de lagen op basis hiervan aan. Deze stap heet *backpropagation*.

Een cyclus van *forward* en *backpropagation* van alle trainingsdata wordt een *epoche* (Engels: *epoch*) genoemd. In de regel wordt de gehele dataset opgedeeld in kleinere *batches*, en worden de gewichten aangepast nadat de foutmarge van zo'n hele batch is berekend. Het doorlopen van één batch heet een *iteratie* (Engels: *iteration*). Dus als je trainingsdata bestaat uit 10 datapunten en je een batch-grootte hebt van 2, dan heb je 5 iteraties nodig voor een epoche. De vraag naar verhouding tussen de batch-grootte en het aantal iteraties per epoche heeft vooral te maken met de leersnelheid en de hoeveelheid verbruikt geheugen; ook dit is weer onderdeel van de *hyperparameter tuning*.

8.2 De kostenfunctie van neurale netwerken

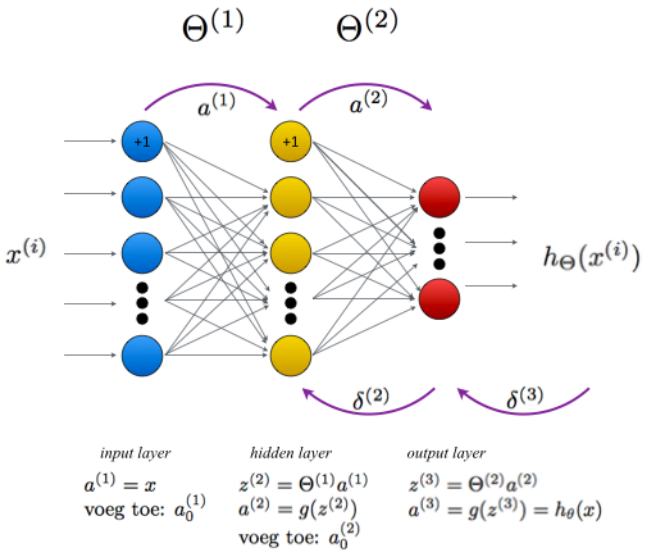
Het idee van een neuraal netwerk is dat de foutmarge na een aantal epochen dusdanig klein is geworden dat we het kunnen gebruiken om een voorspelling te doen over de classificatie van *nieuwe* data. Voor het berekenen van de fout van het volledige netwerk maken we opnieuw gebruik van de kostenfunctie. Omdat we nu niet te maken hebben met één enkele output (wat bij binaire classificatie wel het geval is), moeten we het totaal berekenen van alle nodes in de output-laag. Voor elke node k in de output-laag gebruiken we de formule van

de logistische regressie. Als we in deze laag K nodes hebben (dus we willen de data classificeren in één van K categorieën), dan is de kost die een voorspelling heeft met huidige waarden van Θ gegeven door

$$J(\Theta) = -\frac{1}{m}\left[\sum_{k=1}^{m}\sum_{j=1}^{K} y_k^{(i)} log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) log(1 - (h_\Theta(x^{(i)}))_k)\right].$$

Wanneer we nu één voorbeeld (x, y) nemen om de $h_\Theta(x)$ in figuur 18 uit te rekenen, gebruiken we voor de *forward propagation* het volgende stappenplan:

1. $a^{(1)} = x$
2. voeg toe: $a_0^{(1)} = 1$
3. bereken: $z^{(2)} = \Theta^{(1)} \cdot a^{(1)}$
4. bereken: $a^{(2)} = g(z^{(2)})$
5. voeg toe: $a_0^{(2)} = 1$
6. bereken: $z^{(3)} = \Theta^{(2)} \cdot a^{(2)}$
7. bereken: $a^{(3)} = g(z^{(3)}) = h_\Theta(x)$



Figuur 18: Het stappenplan van forward propagation.

Deze $h_\Theta(x)$ gebruiken we dan om $J(\Theta)$ uit te rekenen (onthoud dat $h_\Theta(x)$ een $K \times 1$ kolomvector is). Aan de hand van de rijvector y, kunnen we de

foutmarge in de output-laag berekenen. Deze foutmarge gebruiken we om de fout van de *individuele* gewichten in de matrices $\Theta^{(2)}$ en $\Theta^{(1)}$ te berekenen (let op dat we hier te maken hebben met een *scalaire* waarde, dus de operaties op de matrices is *element wise*). Het stappenplan voor de *backpropagation* is dan als volgt:

1. bereken: $\delta^{(3)} = a^{(3)} - y$
2. bereken: $\delta^{(2)} = \Theta(2) \cdot \delta^{(3)} \times (g'(z^{(2)})$ (*element wise*)
3. update: $\Theta^{(2)} := \Theta^{(2)} + a^{(2)} \cdot \delta(3)$
4. update: $\Theta^{(1)} := \Theta^{(1)} + a^{(1)} \cdot \delta(2)$

9 De Confusion Matrix

Nu we hebben besproken hoe wel een classificeerder of een neuraal netwerk op basis van trainingsdata kunnen trainen, rijst natuurlijk de vraag *hoe goed* dit netwerk werkt voor *nieuwe* observaties. Uiteindelijk is immers het doel om op basis van bekende data voorspellingen te doen over nog onbekende data. Een eerste naïeve benadering van dit vraagstuk is eenvoudig te kijken hoeveel procent van de data correct wordt geclassificeerd. Wanneer we bijvoorbeeld kijken naar het netwerk dat onze e-mails in vier categorieën moet verdelen, kunnen we tellen hoe vaak dit een spam-bericht correct als spam-bericht classificeert, en dit delen door het totaal aantal berichten.

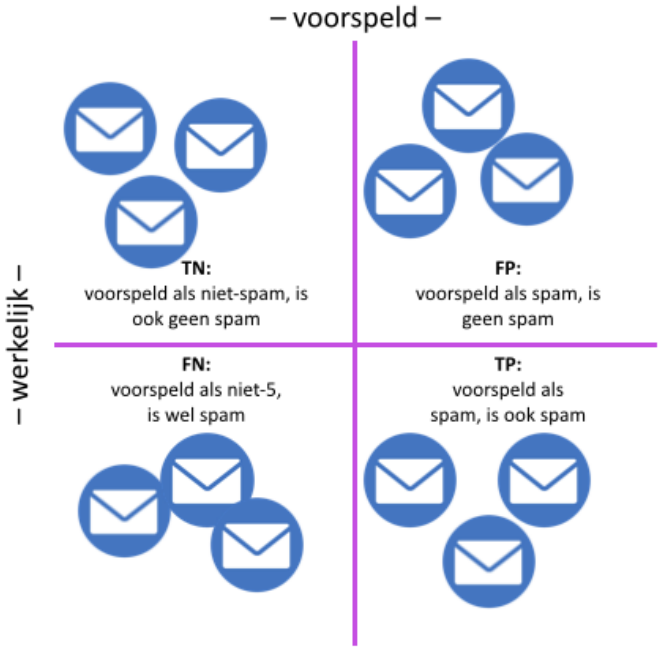
Het probleem met deze aanpak is dat het niet zo gek veel zegt. Als je in plaats van een netwerk een programma schrijft dat gewoon standaard teruggeeft dat het bericht in kwestie een spam-bericht is, is het statistisch waarschijnlijk dat dit in een kwart van de gevallen correct is. Omgekeerd, een programma dat ongeacht de input aangeeft dat dit bericht *geen* spam is, is in driekwart van de gevallen correct. Hoe meer klassen we in de data in willen verdelen, hoe nijpender dit probleem wordt.

Een betere manier om de performance van een classificeerder (of dit nu een neuraal netwerk is of niet) te onderzoeken is door het opstellen van een zogenaamde *confusion matrix*. Hierbij kijk je hoe vaak een observatie van klasse A geclassificeerd is als klasse B; dit totaal zet je in een matrix, waarbij de *rijen* corresponderen met de *werkelijke* waarden, en de *kolommen* corresponderen met de *voorspelde* waarden.

Laten we als voorbeeld even kijken naar e-mailberichten waarvan door een classificeerder bepaald moet worden of een bericht spam is of niet. De vier mogelijkheden die zich hier voordoen, zijn weergegeven in figuur 19. De bovenste regel in deze figuur bevat alle berichten die *geen* spam zijn, de onderste regel bevat de berichten die dit wel zijn. De linkerkolom bevat de berichten waarvan onze classificeerder vindt dat het *geen* spamberichten zijn, de rechterkolom bevat de berichten die worden geclassificeerd als spam.

Berichten die correct zijn geclassificeerd als niet-spam, noemen we *terechte negatieven* (Engels: *true negative*); berichten die zijn geclassificeerd als niet-spam, maar wel spam zijn heten *foutnegatieven* (Engels: *false negative*). Omgekeerd noemen we berichten die werkelijk spam zijn en ook als zodanig zijn geclassificeerd *terechte positieven* (Engels: *true positive*) en berichten die als spam zijn geclassificeerd maar dat niet zijn *foutpositieven* (Engels: *false positive*). In de matrix in figuur 19, en in de berekeningen hieronder, worden deze waarden aangeduid met TN, FN, TP en FP, respectievelijk.

Een perfecte classificeerder zou alleen terechte positieven en terechte negatieven hebben, zonder fouten. De confusion matrix van zo'n classificeerder zou dus gelijk zijn aan een diagonaalmatrix – alleen niet-nul waarden wanneer $i \neq j$. Voor de nagenoeg alle classificeerders is dit echter een onhaalbare kaart.



Figuur 19: Een voorbeeld van een confusion matrix.

9.1 Precisie en sensitiviteit

Doordat we in de confusion matrix de correcte en de incorrecte voorspellingen in één overzicht hebben, kunnen we hier veel informatie uit destilleren. Niet alleen de individuele cijfers, maar juist de *verhouding* hiertussen is van belang. Zo kunnen we iets zeggen over de accuratesse van de classificeerder door te kijken naar hoeveel van de voorspelde waarden ($TP + FP$) ook daadwerkelijk die waarde hebben (TP). Deze metriek heet de *precisie* (Engels: *precision*, ook wel *positive predictive value* of PPV):

$$PPV = \frac{TP}{TP + FP}$$

Je kunt natuurlijk een PPV hebben van honderd procent verkrijgen door je te focussen op slechts één positieve voorspelling en je ervan te vergewissen dat deze correct is ($1/(1+0) = 1$). Een dergelijke classificeerder is echter niet zo heel nuttig, omdat hij alle verdere observaties negeert. Hierom wordt de precisie in de regel gecombineerd met de *sensitiviteit* (Engels: *sensitivity*, ook wel de *recall* of de *true positive rate* of TPR): de verhouding tussen de correct geclassificeerde observaties en het totaal aantal observaties dat werkelijk tot die klasse behoort:

$$TPR = \frac{TP}{TP + FN}$$

Het is vaak handig om de verhouding tussen de precisie en de sensitiviteit in één waarde uit te kunnen drukken, bijvoorbeeld wanneer je meerdere classificeerders eenvoudig met elkaar wilt vergelijken. Hiervoor wordt de zogenaamde F1-score gebruikt: het harmonisch gemiddelde van de precisie en de sensitiviteit.

$$\begin{aligned} F_1 &= 2 \times \frac{\text{precisie} \times \text{sensitiviteit}}{\text{precisie} + \text{sensitiviteit}} \\ &= \frac{TP}{TP + \frac{FN+FP}{2}} \end{aligned}$$

Zoals je kunt nagaan is de F1 score beter naarmate de PPV en de TPR dichter bij elkaar komen te liggen. Er bestaat echter een afruil tussen de PPV en de TPR: je kunt geen classificeerder maken die zowel een hoge PPV als een hoge TPR heeft. Welke van deze twee je belangrijker vindt, is een keuze die samenhangt met het domein en de functie van je systeem. Als je bijvoorbeeld een netwerk wilt trainen dat video's van dierenmishandeling opzoekt, is de sensitiviteit belangrijker dan de precisie – je krijgt dan wel een paar keer onterecht een melding van een foute video (lage precisie), maar je haalt wel de meeste daadwerkelijk foute video's eruit (hoge sensitiviteit). Omgekeerd kun je een netwerk maken dat bepaalt of een video geschikt is voor jonge kijkers. In dat geval heb je waarschijnlijk liever een systeem dat af en toe geschikte video's afkeurt (lage sensitiviteit), zolang het maar zoveel mogelijk slechte video's filtert (hoge precisie).

9.2 Andere waarden

Van belang zijn verder nog de *negatief voorspellende waarde*, de *foutnegatieve verhouding* en de *foutposieve verhouding*. Net als de positief voorspellende waarde zegt de negatief voorspellende waarde (Engels: NPV) iets over de verhouding tussen de voorspelde negatieve waarden en het totaal aantal werkelijke negatieve waarden. Ook de keuze of je de PPV of de NPV zwaarder vindt wegen hangt af van het domein en het doel van je systeem. Is de positieve voorspellende waarde belangrijker dan de negatieve, of juist andersom? Bijvoorbeeld in het geval van ziekte-indicaties kan dit veel uitmaken.

De foutnegatieve en foutpositieve verhoudingen (Engels: *False Negative Rate*, FNR en *False Positive Rate*, FPR, respectievelijk) zijn eenvoudig de waarden van de foutief geclassificeerde observaties ten opzicht van het totaal.

$$FNR = \frac{FN}{FN + TP}$$
$$FPR = \frac{FP}{FP + TN}$$

Hoewel er nog meer verhoudingen en metrieken uit de confusion matrix te identificeren zijn, volstaan degene die hierboven besproken zijn wel om een goede en objectieve vergelijking tussen verschillende systemen en classificeerders mogelijk te maken.