

# En kort introduktion til Maskinlæring; Diabetes — er en person syg eller rask?

STEMAcademyML

November 2025

I fysik vil vi ofte gerne bruge eksisterende data til at forudsige ting. Hvis vi har rigtig meget forskelligt data, kan det være svært for os som mennesker at overskue. Her kan vi bruge maskinlæring og kunstig intelligens til at hjælpe os. Det kan ske både inden for klassifikation af himmellegemer, gletsjerstørrelser og partikler i partikelfysik. Her begynder vi dog med et lidt mere håndgræbligt eksempel - Kan vi forudsige om en person har en sygdom eller ej?

## 1 Introduktion

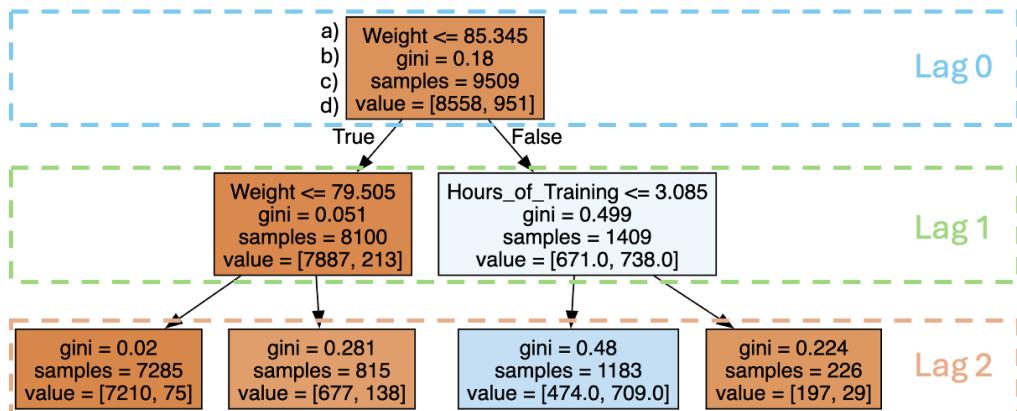
Du har landet dit første studiejob hos Big Pharma ApS. Du er ansat som assistent, blandt andet til at holde styr på en masse data som firmaet har indsamlet. CEO Lars har en ide om at den data kan bruges til at rette annoncer til personer som kunne have gavn af Big Pharmas vitaminpiller. Men GDPR står i vejen for at få fat i personers sygdomshistorik, så vi skal finde en anden måde at finde de personer vi skal rette vores annoncer til. Heldigvis har vi nogle spørgeskemaer som nogle af vores kunder har udfyldt, med informationer omkring deres vægt, hvor mange minutter de dyrker motion, deres blodtryk m.m. og i en af undersøgelserne svarede de også på om de havde diabetes eller ej, før Big Pharma fik at vide at de ikke måtte spørge om det. Men dem vi har indsamlet, kan vi bruge til at forudsige om de andre har, og det er her machine learning træder ind.

Vores mål er at bygge en funktion, der kan tage mange forskellige oplysninger om en person, for eksempel alder, vægt og blodtryk, og ud fra dem forudsige om en person har diabetes eller ej. At skrive sådan en funktion direkte ville være ekstremt svært (eller rettere sagt, *umuligt*). I stedet kan vi lade computeren selv lære sammenhængene ud fra eksempler.

I fysik bruges machine learning ofte til 2 forskellige formål. Den ene, **Classification**, bruges til at afgøre, hvilken kategori noget tilhører, fx om en person er syg eller rask. Den anden, **Regression**, bruges til at forudsige en konkret værdi, fx en persons alder eller insulintal. I dette eksempel vil vi kigge på classification, da CEO Lars gerne vil have os til at inddale personer i to grupper, om de har diabetes eller ej.

## 2 Decision trees

Et **Decision Tree** er bygget op af lag og grene. Ved hver gren stiller modellen et spørgsmål, og bevæger sig ned i det næste lag baseret på om spørgsmålet er sandt eller falsk. Og ved at lære af en masse data, kan den finde ud af hvilke spørgsmål der er bedst at stille.



Figur 1: Et decision tree med to lag. I hver boks ser vi: a) hvilket spørgsmål der stilles, b) gini koeficienten, der beskriver hvor 'blandet' dataen i boksen er, c) hvor mange personer eller hvor meget data der er i gruppen, d) Hvor mange personer der er i hver kategori [rask, syge].

Her bruges et mål som **Gini impurity** der måler, hvor blandet en gruppe er. Jo lavere tallet er, jo bedre har træet lært at adskille klasserne. For eksempel betyder et gini tal på 0.5 at der er lige mange med og uden diabetes i gruppen, så stort set ingen adskillelse, hvorimod et gini tal på 0.0, betyder at der kun er en slags i gruppen, og spørgsmålet er en god indikator.

Hvis algoritmen ikke får lov til at stille nogen spørgsmål, kan den kun gætte på en ting, nemlig det der er mest af. I tilfældet her ville den sige at alle er raske, og det svar ville faktisk være 90 % korrekt da kun 951 personer i dataen har diabetes, mens 8558 er raske, men algoritmen ville være ubrugelig til at opdage diabetes. Dette er en typisk fælde i classification.

Giver vi den lov til at stille et spørgsmål, i dette tilfælde om en persons vægt er mindre end 85 kg, får vi 2 nye grupper. Her går det lidt bedre. Gruppen til venstre har flest raske, men der er stadig 213 personer der bliver vurderet raske som har diabetes - det kalder vi **false negative**. Til venstre vil personerne bliver grupperet som diabetikere, men her er faktisk 671 raske - det kalder vi **false positive**. Men ved igen at stille flere spørgsmål, kan vi minimere andelen af false positive og false negative. Alt efter situationen kan det være bedst at fokusere på at minimere false negative eller false positive. I medicin vil vi typisk hellere vurdere nogle raske som værende syge, end at vurdere nogle syge som raske, der derved ikke får behandling.

## 2.1 Optimering: Boosted decision tree (BDT)

Data kan være meget komplekst, med mange variabler og stor variation i værdier. Det gør det svært at lave ét træ, der rammer godt. Man skal fx bestemme hvor dybt træet må være, hvor mange personer der højest/mindst må være i en kasse, og hvor store fejl der er acceptable. På fagsprog kaldes det hyperparametre.

I stedet for at satse på ét perfekt træ, kan man lade algoritmen bygge mange små træer, hvor hvert nyt træ fokuserer på de fejl, de tidligere træer lavede. På den måde forbedres modellen trin for trin. Denne metode kaldes **Boosted Decision Trees**, og den giver typisk meget mere præcise forudsigelser end ét enkelt træ.

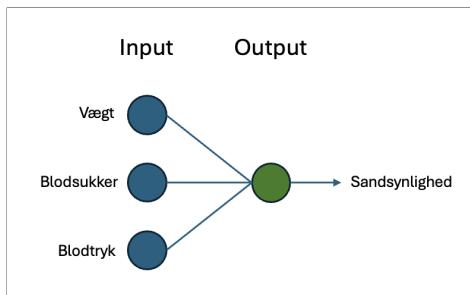
## 3 Neurale Netværk

En anden type algoritme, som man ofte hører om, er **Neurale Netværk (NN)**. Navnet kommer fra, at opbygningen minder om måden neuroner i hjernen sender signaler på. Ligesom et decision tree består af lag, har NN også lag af noget vi kalder **noder**. I stedet for ja/nej-spørgsmål, fungerer de som justerbare knapper. Hver node har en **vægt**, der bestemmer, hvor meget et input betyder for resultatet.

### 3.1 Simpelt NN

Lad os tage et helt simpelt netværk, med kun 1 node. Som nævnt i indledningen, vil vi gerne skrive en funktion, der kan fortælle os om en person er syg, og det kan et NN hjælpe os med. Hvis vi for eksempel har 3 variable, for eksempel vægt, blodsukker og blodtryk, bliver der i noden lavet ligningen:

$$a \cdot \text{vægt} + b \cdot \text{blodsukker} + c \cdot \text{blodtryk} + d = \text{værdi}. \quad (1)$$



Figur 2: Et simpelt neuralt netværk, hvor vi har 3 input og en enkelt node. I noden laves ligning (1), hvor hver inputvariabel ganges med en vægt og lægges sammen plus en bias. Outputtet fra noden er en sandsynlighed for om personen har diabetes eller ej.

Funktionen giver os en værdi som kan indikere om en person har diabetes eller ej.  $a$ ,  $b$  og  $c$  **vægte**. En høj vægt, betyder at variablen er vigtig.  $d$  er en **bias**, der fungerer som et startpunkt (ligesom  $b$  i  $y = ax + b$ ). Bias gør, at modellen ikke behøver gå gennem nul og dermed kan lave mere

realistiske forudsigelser. Men ligningen her kan give os mange værdier, ikke kun mellem 0 og 1. Så vi skal konvertere de tal vi får ud, til en sandsynlighed. Det gør vi med en **activation function**.

For at få en sandsynlighed som output, vil vi ofte bruge en activation function kaldet **Sigmoid**. Den kan tage en hvilken som helst værdi, og omsætte det til en værdi mellem 0 og 1. Hvis vi kalder vores ligning (1) for  $g$ , vil den endelige ligning være

$$\frac{1}{1 + e^{-g}} = \text{sandsynlighed.} \quad (2)$$

og vi får et tal mellem 0 og 1 ud. Jo større  $g$  er, jo tættere på 1 vil sigmoid funktionen være.

NN arbejder bedst med værdier mellem 0 og 1, så derfor er det vigtigt at dataen først bliver normaliseret. Det vil sige hvis vi havde data på personers vægt fra 50 til 150, ville alle værdier blive divideret med 150, så den største vægt ville være 1, og alle andre ville have værdier mellem 0 og 1.

Hvis vi tog et netværk med kun 1 node og vi havde en person på 100 kg, et blodsukker tal på 7 mmol/l og et blodtryk på 110, kunne ligningen være

$$0.9 \cdot 0.6 + 0.7 \cdot 0.9 + 0.1 \cdot 0.5 + 0.1 = 1.32 \quad (3)$$

$$\frac{1}{1 + e^{-1.32}} = 0.79. \quad (4)$$

a er for eksempel høj, fordi høj vægt kan lede til diabetes, og c er lille fordi blodtryk ofte har meget lille betydning for diabetes. Og når vi bruger vores sigmoid, får vi en sandsynlighed på 79 %. Hvis vi sagde at alle med over 50 % chance ville klassificeres som diabetikere, ville denne person altså blive klassificeret som syg.

### 3.2 NN med flere lag

Det er en god start, men, ofte vil vi se at der er mange ting at tage højde for, der kan være sammenhænge som for eksempel at blodsukker har stor betydning hvis man er overvægtig, og mindre betydning hvis man er normalvægtig, eller at vægt er mindre betydningsfuldt hvis personen dyrker meget motion. Disse sammenhænge kan vi ikke fange, hvis vi kun har en ligning.

Det er her vi introducerer det der kaldes **"Hidden layers"**. Et hidden layer er et lag af noder, der ligger mellem input og output (se figur 3). Hver node i "hidden layer", laver samme ligning som (1), med de input den får, og sender så et tal videre til det næste lag. Men inden et tal sendes videre til det næste lag, bruger vi igen en **activation function**, men her har activation function et andet formål. Det er at aktivere forskellige noder i modsætning til i output, hvor dette var at omdanne det endelige tal til en sandsynlighed.

Her vil man ofte bruge den activation function der hedder **ReLU** (Rectified Linear Unit):

$$f(x) = \max(0, x) \quad (5)$$

Hvis ligning (1) giver en negativ værdi, vil  $f(x)$  sende 0 videre til næste lag, mens positive værdier sendes videre som de er. Det betyder, at nogle noder "slukker", og andre "tænder", afhængigt af dataen. På den måde kan forskellige noder lære at reagere på forskellige typer af personer — fx personer med overvægt og personer der dyrker meget sport. På den måde kan vi for eksempel have en høj vægt på personers vægt når de dyrker lidt motion, men en lavere vægt hvis de dyrker meget motion.

### 3.3 Opbygning af NN

Et neutralt netværk kan for eksempel se sådan ud som i figur 3.

Her har vi 3 input parametre. Herefter to hidden layers, hvor ligningerne som (1) og (2) bliver udført med en ReLU activation function, og til sidst et output lag, der giver den endelige sandsynlighed med en sigmoid activation function.

Hvis et hidden layer har  $n$  input (noder i det foregående lag) og  $m$  noder i sit eget lag, har det

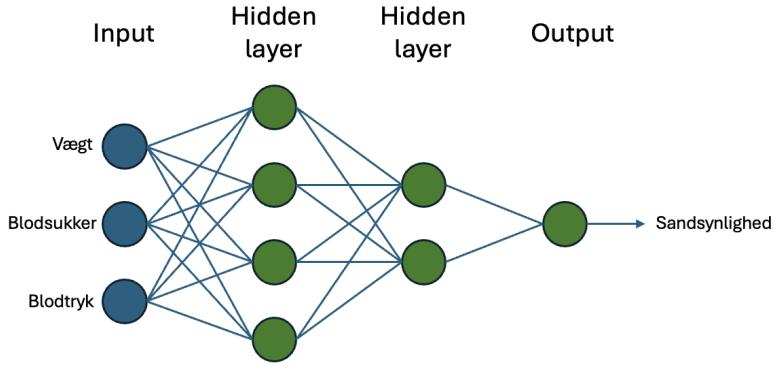
$$n \times m + m$$

parametre.

Så for første skjulte lag har vi  $3 \times 4 + 4 = 16$  parametre, for andet skjulte lag  $4 \times 2 + 2 = 10$ , og for output laget  $2 \times 1 + 1 = 3$ . I alt altså 29 parametre. Vi kan også sige, at noderne svarer til antallet af **bias**, mens stregerne imellem svarer til antallet af **vægte**.

Til sidst justerer algoritmen alle vægte og biaser, så forskellen mellem den forudsagte sandsynlighed for, at en person er syg, og den faktiske sandhed bliver så lille som muligt. Det betyder, at modellen hele tiden prøver at blive bedre til at ramme de rigtige svar.

På den måde lærer et neuralt netværk selv at finde mønstre i data, der kan være så komplekse, at vi aldrig ville beskrive dem med en enkelt formel.



Figur 3: Et eksempel på et neuralt netværk med 3 input parametre og to skjulte lag og et output. I hver node laves ligning (1). I noderne i ”Hidden Layer” bruger vi typisk ReLU activation function før resultatet sendes videre til næste node, og i den sidste output node bruger vi typisk Sigmoid activation function for at lave det om til en sandsynlighed.

## 4 Hvordan ved algoritmen hvilket og hvor mange spørgsmål den skal stille?

En algoritme skal bruge noget til at måle, hvor godt den gør det, så den ved, hvilke spørgsmål den skal stille, eller hvilke vægte den skal give de forskellige parametre. I Machine Learning kalder vi det en **Loss function**.

For et boosted decision tree prøver algoritmen mange mulige spørgsmål og beregner, hvor meget hvert spørgsmål kan reducere loss function. Det spørgsmål, der giver det mindste bidrag, bliver valgt. Processen gentages i hver ny gren, indtil trætet er ”dybt nok” eller ikke længere forbedres. Bemærk, for et simpelt decision tree bruger man den ovennævnte ginikoefficient.

For et neuralt netværk fungerer principippet på samme måde, men i stedet for at vælge spørgsmål prøver algoritmen forskellige vægtkombinationer i de mange noder. Den beregner herefter sin loss (fejl), og justerer vægtene lidt ad gangen, så fejlen bliver mindre for hver iteration. Denne proces er det vi kalder **træning**.

Den hyppigst brugte loss function for classification er **Log Loss** (eller Binary Cross Entropy):

$$\mathcal{L}(\hat{y}, y) = -\frac{1}{N} \sum_{n=1}^N [y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \quad (6)$$

hvor  $y$  er den sande værdi, som er enten 0 eller 1, alt efter om personen har diabetes eller ej og  $\hat{y}$  er sandsynligheden mellem 0 og 1 fra vores algoritme. Jo længere væk sandsynligheden er fra den sande værdi, jo større er vores loss function. Algoritmen bliver ved med at justere dens parametre til loss function er så lille som mulig.

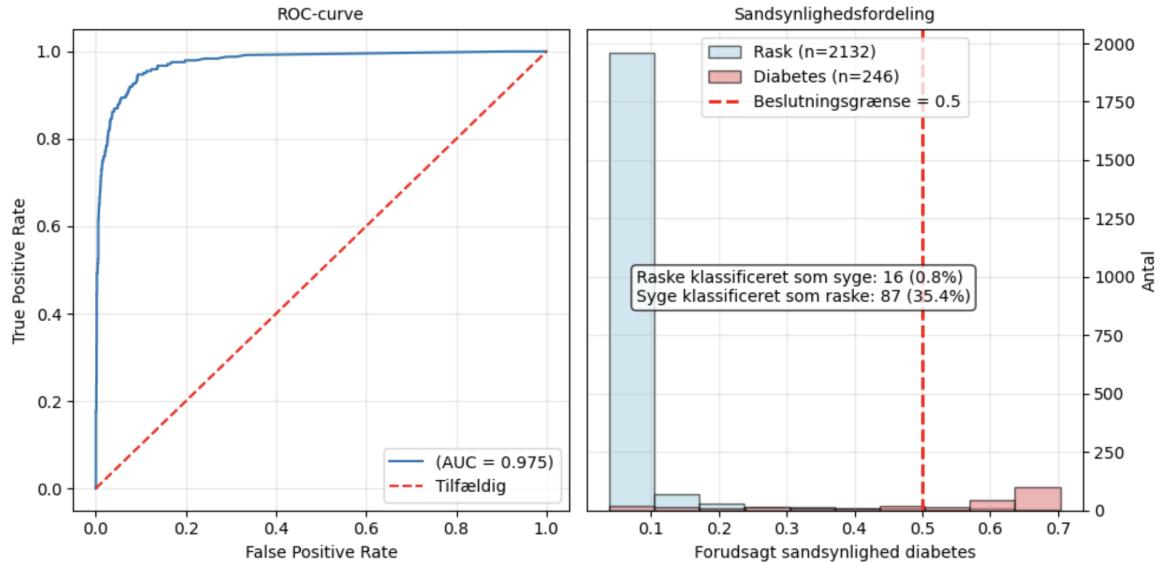
### 4.1 Hvordan kan vi se vi hvor godt algoritmen klarer sig?

Når nu vi har fået opbygget vores decision tree eller neurale netværk, vil vi gerne finde ud af hvor god den er til at forudsige om en person har diabetes eller ej. Det gør vi ved at give modellen ny data, som den ikke har trænet på, og bede den om at forudsige, og så sammenligne med de rigtige værdier. Typisk deler man datasættet i to dele: en del til **træning**, hvor modellen får lov til at lære

sammenhængene, og en mindre del til **test**, som gemmes væk, indtil modellen skal vurderes. På den måde sikrer vi, at modellen ikke bare husker de præcise svar, men faktisk har lært et mønster, der også virker på ny data.

For classification vil man typisk visualisere performance med en **ROC kurve** (Receiver Operator Characteristic), som vist til venstre i figur 4. Den viser forholdet mellem false positive rate og true positive rate. Vi vil gerne have denne til at ligge så tæt på det venstre øverste hjørne som muligt. **AUC-scoren** (Area Under Curve) i hjørnet skal være så tæt på 1 som muligt. En AUC på 0.5 ville svare til at gætte tilfældigt.

Til højre er **sandsynlighedesfordelingen** plottet. Hvis modellen er god, ligger de raske mest omkring lave sandsynligheder (tæt på 0), og de syge ligger mest omkring høje sandsynligheder (tæt på 1). Jo mere de to fordelinger er adskilt, jo nemmere er det at skelne mellem syg og rask. Alt til højre for beslutningsgrænsen bliver klassificeret som "syg", og alt til venstre som "rask". Hvis fordelingerne er godt adskilt, betyder det, at vi kan vælge en grænse, der laver meget få fejl — altså få raske, der bliver kaldt syge, og få syge, der bliver kaldt raske.



Figur 4: Til vestre ses en ROC-curve, og den tilhørende sandsynlighedsfordeling til højre. Når vi deler ved 0.5, er der kun 0.8 % raske der klassificeres som syge, men 35 % syge bliver klassificeret som raske, hvilket er hvad vi gerne vil minimere. Den er stadig meget bedre end et tilfældigt gæt som ses ved at ROC-kurven er tæt på venstre hjørne.

## 5 Er alle informationer lige vægtige?

Når vi har trænet vores model, er det interessant at finde ud af, hvilke variable den lægger mest vægt på. I et **decision tree** kan vi ofte allerede se det ud fra de spørgsmål, træet stiller først, for eksempel at kvadratmeter og postnummer har stor betydning for salgsprisen. I et **neuralt netværk** er det sværere at se direkte, men her kan vi kigge på, hvor meget de enkelte vægte påvirker resultatet.

En mere generel metode, der virker for næsten alle typer modeller, kaldes **Permutation Importance**. Her tager man den trænede model og "roder" med én variabel ad gangen. Det vil sige, man blander dens værdier tilfældigt, så den mister sin sammenhæng med resten af dataen. Hvis modellens præcision falder meget, betyder det, at den variabel var vigtig for forudsigelsen. Hvis vi fx bytter rundt på personernes alder, og modellens præcision falder markant, betyder det, at alder er vigtig. Hvis der næsten ingen forskel er, betyder det, at modellen ikke brugte den variabel særlig meget.

Det fantastiske ved den her metode er, at vi ikke behøver at vide noget som helst om systemet på forhånd. Modellen finder helt selv ud af, hvilke ting der betyder mest, og det kan være mønstre, som folk normalt kun opdager efter mange års erfaring. Pludselig kan vi se, hvilke parametre der betyder noget for om man har diabetes eller ej, uden at vide noget som helst om sygdommen eller bare medicin generelt. Det er her, Machine Learning virkelig bliver spændende: når den lærer os noget nyt om verden, som vi ikke selv havde set.