En kort introduktion til Maskinlæring; Hvad koster et hus?

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, gletsjer størrelser og partikler i partikelfysik. Her begynder vi dog med et hverdagseksempel - Kan vi gætte prisen på et hus?

1 Introduktion

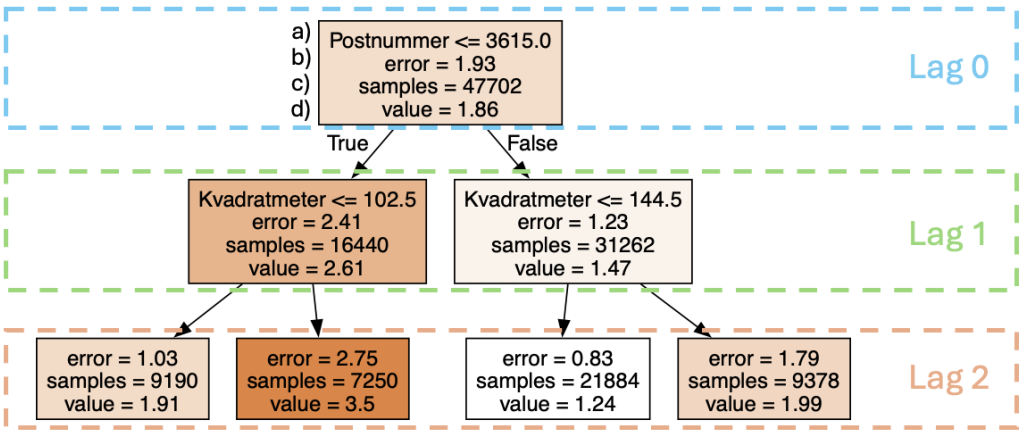
Martin har lige startet sin første ejendomsmægler-virksomhed. Han har ikke nogen erfaring, og det kan tage mange år at få en god fornemmelse for hvad husene skal prissættes til. Heldigvis har han fået adgang til en database over huse som der tidligere er solgt. I databasen er der oplysninger såsom husets størrelse, postnummer, hvor langt der er til den nærmeste skole m.m., og hvilken pris huset blev solgt for. Men han gider ikke sidde og studere alle de tidligere cases. I stedet har han hørt at man kan lave en machine learning algoritme, der kan sige hvad husene skal koste. Han har aldrig lavet machine learning før, men hvor svært kan det være? Og på rigtig direktørmanér, har han besluttet at uddelegere opgaven, til sin nye billigt ansatte studentermedhjælper, dig.

Vores mål er at bygge en funktion, der kan tage mange forskellige oplysninger om et hus, for eksempel placering, størrelse og husnummer, og ud fra dem forudsige, hvad huset bør koste. 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 bolig er et hus eller en lejlighed. Den anden, **Regression**, bruges til at forudsige en konkret værdi, fx salgsprisen på et hus. Martin vil gerne have os til at forudsige konkrete salgspriser, så derfor arbejder vi med regression her.

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. Priserne er i millioner. Hver boks repræsenterer en gruppe af huse og viser: a) hvilket spørgsmål (også kaldet **split**) der bedst opdeler husene i gruppen, b) hvor stor fejl (**loss** eller squared error) der er, hvis man gætter gennemsnitsprisen for gruppen (læses 1.9 mio.), også kendt som usikkerhed, c) hvor mange huse der hører til gruppen, og d) hvad den gennemsnitlige salgspris er for husene i gruppen.

Ofte hører man udtrykket, at en model skal **trænes**. Det betyder, at vi viser modellen en masse eksempler, hvor vi allerede kender det rigtige svar — i vores tilfælde husets salgspris. Ud fra de eksempler lærer modellen, hvilke spørgsmål der er gode at stille, ved hele tiden at sammenligne sine egne gæt med de rigtige svar. Når den har øvet sig på den måde, kan den bruge det, den har lært, på ny data, hvor svaret ikke er kendt, og give et kvalificeret bud på prisen.

Hvis algoritmen ikke får lov til at stille nogen spørgsmål, kan den kun gætte på gennemsnitsprisen for alle huse i datasættet som i dette tilfælde er ca. 1.8 mio. kr. Det er en meget simpel model, der altid vil gætte det samme tal.

Giver vi den lov til ét lag, kan den stille ét spørgsmål og dele data i to grupper. I vores tilfælde stiller algoritmen spørgsmålet: "Er postnummeret mindre end 3615?". Hvis postnummeret er mindre, gætter den på 2.6 mio, og er det større, gætter den på 1.4 mio. Algoritmen prøver sig frem med en masse forskellige spørgsmål, og vælger det der giver den bedste opdeling, det der gør at prisen er tættest på den rigtige pris.

Med to lag kan træet stille et nyt spørgsmål i hver gruppe. I dette tilfælde spørger den om kvadratmeter, størrelsen på huset. Og alt efter hvilket postnumre, giver det bedst mening at dele ved forskellige størrelser. Er vi tættere på København har det stor betydning om det er større eller mindre end $\approx 100\ m^2$, men for de større postnummer, giver det mere mening at dele ved $\approx 150\ m^2$. Nu har træet fire mulige gennemsnitspriser at gætte ud fra, og præcisionen forbedres.

2.1 Optimering: Boosted decision tree (BDT)

Data kan være meget komplekst, med mange variabler og stor variation i værdier. Vi kunne i princippet lave et træ, der perfekt kunne forudsige alle priser i vores datasæt. dvs. hvis vi havde 10.000 huse, kunne vi bare lave træet dybt nok til at det kunne adskille alle 10.000 huse, og gætte deres specifikke værdi. Men her kan vi falde i den fælde vi kalder **overfitting**. Det vil sige at vores algoritme bliver perfekt tilpasset det ene datasæt, lærer alle de små særtilfælde, og derfor ikke længere er særlig generel. Det svarer til at lære at spille et instrument, ved at perfekte en enkelt sang. Du bliver virkelig god til den ene sang, men har svært ved at spille en ny sang, for du har ikke lært de generelle ting om noder, rytmer og akkorder. Derfor kan det i stedet være en fordel at lave nogle mindre træer, men så lave mange af dem, og kombinere deres svar. Det er det vi kalder **Boosted Decision Trees**.

I praksis foregår det sådan, at modellen trænes i flere runder. Først laver vi det første træ, som kommer med et groft bud på alle huspriserne. Derefter kigger vi på, hvilke huse det ramte dårligt. De huse, hvor fejlen var stor, får nu en større "vægt", som betyder at de bliver vigtigere i næste runde. Når vi så træner træ nummer to, vil det træ altså især prøve at rette de fejl, det første træ lavede. Derefter gør vi det samme igen: vi finder de eksempler, der stadig bliver ramt dårligt, giver dem mere vægt, og lader det tredje træ rette videre. På denne måde lærer hvert nyt træ lidt mere om de svære eller specielle tilfælde, og modellen forbedres trin for trin uden at et enkelt træ behøver at være perfekt.

Det kunne for eksempel være at vi havde 3 **boosting rounds**, hvilket betyder at vi giver algoritmen lov til at lave 3 træer. Det første træ vil komme med et generelt bud, den tager de mere generelle principper i brug som jo tættere på København, jo dyrere, og jo større kvadratmeter, jo dyrere, og kommer med et bud på en pris. Men der er også andre ting der spiller ind, for eksempel, selvom det ikke ligger i København, kan det godt ligge tæt på centrum af en anden by. Det næste træ har måske lært de mønstre, og siger der skal lægges noget oveni prisen. Det 3. træ har så måske lært noget om hvad husets alder har at sige, og kan byde ind med om der skal trækkes noget fra.

Et resultat kunne være

$$Træ_1(x) + Træ_2(x) + Træ_3(x) = salgspris \tag{1}$$

$$1.5mio + 0.5mio - 0.2mio = 1.8mio \tag{2}$$

Som kunne være et større hus i Jylland, der ligger tæt på centrum af Kolding, men er fra 1950. så beliggenheden tæt på en større by i Jylland trækker den generelle pris op, men byggeåret trækker prisen ned.

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 give os en salgspris, og det kan et NN hjælpe os med. Hvis vi for eksempel har 3 variable, for eksempel postnummer, kvadratmeter og husnummer, bliver der i noden lavet ligningen:

$$a \cdot \text{postnummer} + b \cdot \text{kvadratmeter} + c \cdot \text{husnummer} + d = \text{salgspris}. \tag{3}$$

Her er 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. Disse parametre er nogen som modellen lærer, ved at prøve sig frem med forskellige værdier, og se hvad der kommer tættest på den rigtige salgspris.

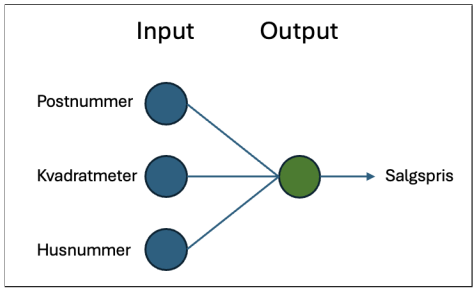
NN arbejder bedst med værdier mellem 0 og 1, så derfor er det vigtigt at dataen først bliver normaliseret. Det vil sige hvis vi havde data med kvadratmeter fra 10 til 100, ville alle værdier blive divideret med 100, så det største hus ville være 1, og alle andre ville have værdier mellem 0 og 1.

Hvis vi tog et netværk med kun 1 node og vi havde et hus på Frederiksberg med postnummer 2000, på 60 m^2 og husnummer 12, kunne ligning (1) blive til:

$$-0.9 \cdot 0.2 + 0.8 \cdot 0.6 + 0.1 \cdot 0.12 + 0.03 = 0.332 \tag{4}$$

og netværket ville helt simpelt være som vi ser i figur 2.

a er for eksempel negativ, fordi ofte er lavere postnumre tættere på København, og derfor dyrere, og c er lille fordi husnummer ofte har meget lille betydning for salgsprisen. Her får vi så en forudsagt salgspris på 3,32 mio. kr[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 modellens bud på salgsprisen.

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, for eksempel vil kvadratmeter have større betydning i byen end på landet. I byen kunne det have stor betydning om lejligheden var 60 eller 80 kvadratmeter, hvorimod på landet, har det måske ikke så stor betydning for prisen om huset er 220 eller 250 kvadratmeter. Så i virkeligheden vil vi gerne have at kvadratmeter har en anden vægt, når postnummeret er lavt.

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, bruges noget der kaldes en **activation function**.

En af de mest brugte activation functions hedder **ReLU** (Rectified Linear Unit):

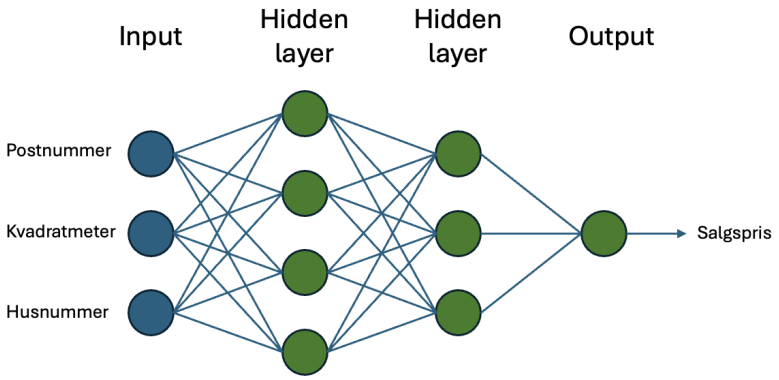
$$f(x) = max(0, x) \tag{5}$$

[1]Det her er et meget konkret eksempel. Hvis det største hus i datasættet for eksempel havde været 350 m^2, så ville vi have haft 60/350 = 0.17 som værdi i stedet, og med en anden normalisering, kunne 0.332 også give noget andet når vi reskalerer det igen.

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 hvilken type hus. På den måde kan forskellige noder lære at reagere på forskellige typer af huse — fx små lejligheder i byen eller store villaer på landet. På den måde kan vi for eksempel have en høj vægt på kvadratmeter når vi har små postnumre og en lav vægt når vi har store postnumre, ved at tænde og slukke for noder med de vægte.

3.3 Opbygning af NN

Et neuralt netværk kan for eksempel se sådan ud:



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), men i noderne i "hidden layer"bliver resultatet også pakket ind i activation function, før det sendes videre.

Her har vi 3 input parametre. Herefter to hidden layers, hvor ligninger som (1) og (2) bliver udført med en activation function, og til sidst et output lag, der giver det endelige bud på salgsprisen.

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 $4 \times 3 + 3 = 15$, og for output laget $3 \times 1 + 1 = 4$. I alt altså 35 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 forudsagte og sande værdier bliver så lille som muligt. På den måde kan neurale netværk finde mønstre, der er alt for komplekse til, at vi selv kunne skrive dem som en enkelt formel.

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 tildele de forskellige parametre. I Machine Learning kalder vi det en **Loss function**.

For et decision tree prøver algoritmen mange mulige spørgsmål og beregner, hvor meget hvert spørgsmål kan reducere fejlen, eller forskellen fra den rigtige slagspris. Det spørgsmål, der giver den største forbedring, bliver valgt. Processen gentages i hver ny gren, indtil træet er "dybt nok"eller ikke længere forbedres.

For et neuralt netværk fungerer princippet 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. Det er denne proces der kaldes træning.

De to mest brugte loss functions for regression er:

- **Absolute error (MAE)**: Her bruges den absolutte forskel mellem forudsagt og sand værdi.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|\text{sande værdi}_i - \text{forudsagte værdi}_i\right|$$

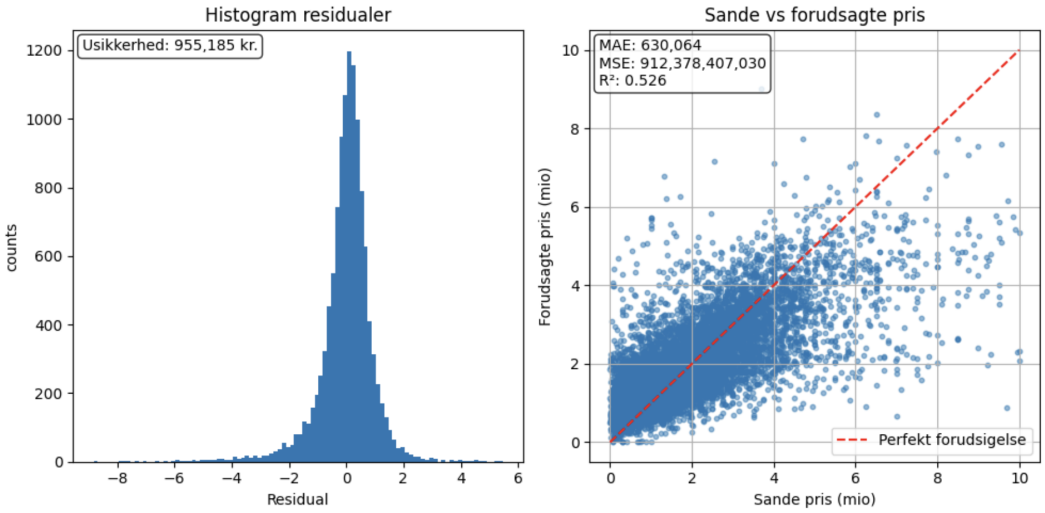
- **Squared error (MSE)**: Her kvadreres forskellen mellem forudsagt og sand værdi. Punkter der ligger langt fra normalen (outliers) bliver straffet hårdere, og derfor bruges MSE ofte når vi har mange outliers.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(\text{sande værdi}_i - \text{forudsagte værdi}_i\right)^2$$

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. 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 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 regression vil man typisk visualisere performance som vist nedenfor: Til venstre er plottet forskellen mellem forudsagte og sande værdier (residualer). Her vil vi gerne have, at punkterne ligger tæt på nul, så forudsigelserne ikke systematisk rammer for højt eller for lavt, og at spredningen er så lav som muligt. Til højre er plottet forudsagte værdier mod sande værdier. Her vil vi gerne have, at punkterne ligger tæt på linjen $y = x$, som betyder, at forudsigelsen og den sande værdi er den samme.



Figur 4: Til venstre ses et histogram over residualerne (forskellen mellem forudsagt og sand pris). De fleste residualer ligger tæt på nul, hvilket betyder, at modellen oftest rammer rigtigt, men der findes også enkelte store afvigelser. Til højre ses de forudsagte priser plottet mod de sande priser. Den røde linje viser en perfekt forudsigelse ($y = x$). Punkterne ligger generelt tæt på linjen, men der er også spredning, især ved de dyreste huse.

5 Er alle informationer lige vigtige?

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, 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 tilfældigt rundt på postnumrene, og modellens præcision falder markant, betyder det, at postnummer er en vigtigt faktor for salgsprisen. 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, hvad der faktisk driver prisen på et hus, eller hvad der gør forskellen mellem to typer af data, uden at have nogen forhåndsviden. Det er her, Machine Learning virkelig bliver spændende: når den lærer os noget nyt om verden, som vi ikke selv havde set.