Forelesningsnotater Introduksjon til Maskinlæring – TDT4172

Inga Strümke

June 25, 2025

Contents

1 Veiledet læring

1.1 Data

Maskinlæring går ut på å programmere datamaskiner slik at de kan lære fra data. Arthur Samuel (1959) ga definisjonen

> *"Machine learning in the field of study that gives computers the ability to learn without being explicitly programmed."*

Tom Mitchell (1997) ga den mer presise definisjonen

> *" A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."*

Utgangspunktet er altså *experience*, representert i form av data, som brukes av et dataprogram hvis *performance*, altså ytelse, øker på en *task*, altså oppgave. Data har vi i form av datasett, som består av datapunkter med ulike egenskaper. Disse egenskapene omtales som *variabler* innen statistikk, og ofte brukes det engelske begrepes *features* i maskinlæringssammenheng. Vi vil bruke disse to begrepene om hverandre. Variabler, eller features, kan være kvalitative eller kvantitative, se eksempler i tabell 1. *Kvalitative variabler*, også omtalt som kategoriske features, kan ha et begrenset antall diskrete verdier, henholdsvis beskrive et endelig antall kategorier. Disse må ofte enkodes før vi kan bruke dem i maskinlæring (for eksempel ved bruk av label encoding), for eksempel hvis de er representert som strenger i stedet for tall. *Kvantitative variabler* beskriver data som kan måles og sorteres. De har numeriske verdier og man kan utføre matematiske operasjoner på dem (addisjon, subtraksjon, osv), og de kan være både diskrete og kontinuerlige. Slike features kan i prinsippet brukes direkte i maskinlæring, men ofte bør de behandles først, for eksempel gjennom normalisering eller mer avansert feature engineering, som vi skal se på senere.

Table 1: To typer features for maskinlæringsmodeller.

	Kvalitative variabler	**Kvantitative variabler**
Representerer	Kategorier	Numeriske verdier
Eksempler	Sjanger {action, komedie, ...}, Aldersgruppe {ung, ..., gammel}, Type frukt {eple, banan, ...}	Inntekt, alder, antall lyttere, areal, nedbørsmengde, temperatur

I et datasett representerer kolonnene ulike features, mens radene inneholder de ulike datapunktene. Når vi gjør maskinlæring ønsker vi å lage en databasert modell som estimerer en funksjonsverdi for gitte data. Hvis datasettet vi bruker inneholder en kolonne som angir funksjonsverdien for det aktuelle datapunktet, omtaler vi denne kolonnen som *target*, og vi kan gjøre veiledet læring, fra engelsk *supervised learning*. Vi kan for eksempel bruke datasettet *Titanic* fra Kaggle[1], hvorav et utvalg er vist i tabell 2. Her representerer PassengerId, Pclass, Name, Sex og Age datasettets fem features, mens Survived representerer target – hvis målet er å lage en modell som beregner sjansen for at en person overlever.

Oppgave:

1. Hvilke variabler er kategoriske?
2. Hvilke variabler er binære?
3. Hvilke variabler er kontinuerlige?

[1] https://www.kaggle.com/c/titanic

Table 2: Titanic-datasettet fra Kaggle.

PassengerId	**Pclass**	**Name**	**Sex**	**Age**	**Survived**
1	3	Braund, Mr. Owen Harris	male	22	0
2	1	Cumings, Mrs. John Bradley	female	38	1
3	3	Heikkinen, Miss. Laina	female	26	1
4	1	Futrelle, Mrs. Jacques Heath	female	35	1
5	3	Allen, Mr. William Henry	male	35	0
6	3	Moran, Mr. James	male		0
7	1	McCarthy, Mr. Timothy J	male	54	0
8	3	Palsson, Master. Gosta Leonard	male	2	0
9	2	Montvila, Rev. Juozas	male	27	1

4. Hvilken variabel bør vi ikke bruke?

Svar på oppgaven:

1. Pclass, Name, Sex, Survived, (PassengerId, om vi anser de ulike Id'ene som hver sin kategori)
2. Sex, Survived
3. Age
4. PassengerId, Name

Grunnen til at vi ikke bør bruker PassengerId, er at denne variabelen ikke inneholder informasjon om egenskapene til en passasjer, og sannsynligvis er tilfeldig valgt. Name bør kun brukes hvis vi ønsker å lage en modell som forstår språk tilstrekkelig godt til å kunne finne en relasjon mellom typiske navn og for eksempel dyktighet i redningsbåter. For modellene vil skal lage, vil ikke Name inneholde relevant informasjon.

Før vi kan begynne på modelleringen, fjerner vi variablene vi ikke vil bruke, og innfører nyttig notasjon, se tabell 3. Hver rad i dataene representerer et datapunkt, og vi bruker indeks i oppe for å angi nummer

Table 3: Titanic-datasettet etter at vi har valgt ut hvilke variable vi vil beholde, og innført notasjon.

	x_1	x_2	x_3	y
$\mathbf{x}^{(1)}$	3	male	22	0
$\mathbf{x}^{(2)}$	1	female	38	1
$\mathbf{x}^{(3)}$	3	female	26	1
$\mathbf{x}^{(4)}$	1	female	35	1
$\mathbf{x}^{(5)}$	3	male	35	0
$\mathbf{x}^{(6)}$	3	male		0
$\mathbf{x}^{(7)}$	1	male	54	0
$\mathbf{x}^{(8)}$	3	male	2	0
$\mathbf{x}^{(9)}$	2	male	27	1

i datasettet. Eventuelt kan vi sette en parentes rundt indeksen, for å ikke forvirre indeksen med en eksponent. Hver kolonne representerer en feature, med indeks i nede. Vi brukes fet $\mathbf{x}$ for å angi en vektor, $\mathbf{x}^{(1)} = (x_1^{(1)}, x_2^{(1)}, x_3^{(1)})$. For å benevne target, altså størrelsen vi ønsker å estimere, bruker vi y.

Vi kan skrive oppgaven vi ønsker å løse som følger

$$\underbrace{(\text{Pclass, Sex, Age})}_{\text{features } \mathbf{x}^{(i)}=(x_1^{(i)},x_2^{(i)},x_3^{(i)})} \rightarrow \underbrace{\text{Survived}}_{\text{target } y^{(i)}} \tag{1}$$

og siden vi har en target per datapunkt, kan vi altså gjøre veiledet læring for å lage en modell som løser denne oppgaven. For å oppsummere:

- Vi har features $\mathbf{x}$ og targets y
- Vi ønsker å estimere y for nye $\mathbf{x}$, basert på relasjonene representert mellom kjente $\mathbf{x}$ og y.

Før vi begynner å lage en modell bør vi få en oversikt over sammenhenger i dataene. **Oppgave:** Se på tabell 3 igjen. Hva bør vi gjøre før vi plotter disse dataene? Svaret lyder:

- Rad 6 mangler en verdi for feature 3; dette vil gi en feilmelding når vi prøver å plotte dataene,
- feature 2 er strenger (strings), altså ikke-numeriske verdier, som en datamaskin ikke kan plotte uten videre,
- feature 3 er av en helt annen størrelsesorden enn features 1 og 2, og kan i verste fall gi misvisende plott.

Vi kan løse disse problemene ved hjelp av innebygget funksjonalitet i `pandas` og `sklearn`. En enkel måte å bli kvitt rader med manglende verdier er med `pandas`-metoden `dropna()`, som fjerner rader der minimum én verdi mangler. For å gjøre om feature 2 til numeriske verdier, kan vi bruke en `LabelEncoder`, som mapper hver feature-verdi til et unikt tall. I vårt tilfelle kan for eksempel "female" mappes til 0, og "male" til 0 (eller omvendt). For å skalere verdiene til feature 3 til intervallet $[0, 1]$, kan vi bruke `MinMaxScaler`. Slike teknikker, der vi fjerner rader, encoder eller skalerer feature-verdier, omtales som preprosessering av dataene.

Etter at vi er ferdige med å redigere dataene, bør vi plotte dem for å se hvordan ulike features fordeler seg. Merk at plotting også kan brukes for å få idéer til hvordan dataene bør behandles, så plotting og dataprosessering kan skje om hverandre. Figur 1 viser fire ulike plott, hvorav figurene 1a og 1b viser bar-plott, og figurene 1c og 1d viser histogrammer, over ulike features og fargekoding basert på verdien til target Survived. Et *bar-plott* viser oss middelverdien per kategori, i dette tilfellet middelverdien til datapunktene som svarer til Survived=1. Vi bruker altså bar-plott for å sammenlikne variablers gjennomsnittsverdi på tvers av kategorier. Et *histogram* viser oss fordelingen av en variabel, i dette tilfellet innad i hver klasse, altså en fordeling for variabelverdier som svarer til Survived=1 og en annen for Survived=0. Stolpene representerer intervaller for variabelens verdi, og disse intervallene omtales som *bins*. Valg av bin-størrelse påvirker hvordan histogrammet ser ut, og det finnes ingen almenngyldig regel for hvor store (hhv hvor mange) bins et histogram bør ha, siden dette er avhengig av dataenes underliggende fordeling. Hvis et histogram ser veldig hakkete ut har man ofte valgt for mange bins, som følgelig inneholder for få datapunkter, og hvis det er vanskelig å få øye på en form i fordelingen, har man ofte valgt for få bins, som følgelig inneholder for mange datapunkter. Kun ved å se på plottene i figur 1 får vi allerede en idé om at passasjerer i Pclass=1, kvinnelige passasjerer og yngre passasjerer ofte overlever. Dette passer godt med det vi vet fra filmen :).

Når vi er ferdige med å preprosessere dataene og har studert dem gjennom plott, er vi klare for å begynne med modelleringen, altså å bygge en modell. For Titanic-dataene ønsker vi å lage en modell f som estimerer sannsynligheten for å overleve, target y, basert på verdiene i features $\mathbf{x}$, altså $p(y|\mathbf{x})$. En modell som gjør dette gir oss *prediksjoner* $f(\mathbf{x}) = y_{\text{pred}}$.

1.2 Logistisk regresjon

Den enkleste modellen vi kan tenke oss er lineær regresjon,

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w}\mathbf{x} + b\,, \tag{2}$$

hvor vektene w_i forteller oss hvor viktig hver feature x_i er for utfallet, altså verdien til z. En nyttig mental øvelse er å tenke over om z er et tall eller en vektor. Dette kan vi se basert på hvordan $\mathbf{w}$ og $\mathbf{x}$, som begge er vektorer, kombineres i uttrykket.

I likning 2 ser vi at z kan bli et hvilket som helst tall, mens vi er ute etter en sannsynlighet for overlevelse, altså et tall mellom 0 og 1. En enkel løsning på dette er å bruke den logistiske funksjonen $\sigma(x)$, også kalt Sigmoid-funksjonen fordi den er formet som en 'S', se figur 2. Denne mapper reelle tall til domenet $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}\,. \tag{3}$$

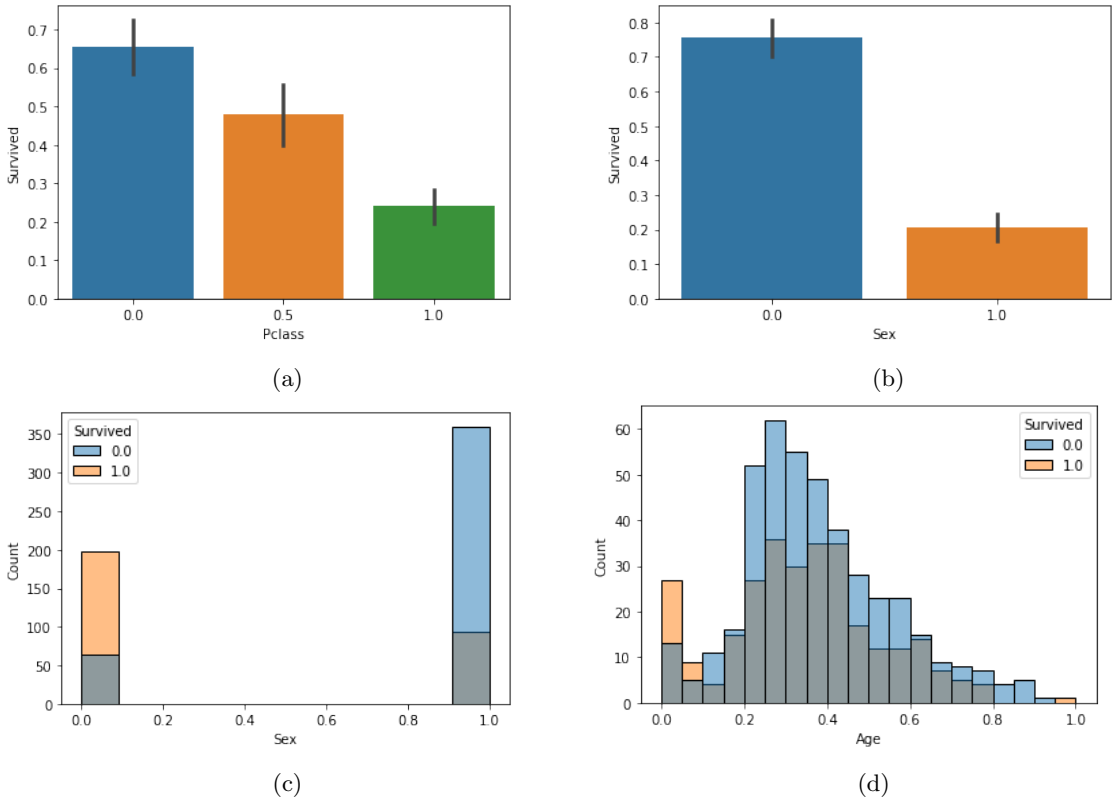


Figure 1: Bar-plot (a, b) og histogrammer (c, d) fra Titanic-dataene. Se beskrivelse i teksten.

Det kan være lurt å huske denne funksjonen, da vi kommer til å møte den igjen når vi diskuterer nevrale nettverk, da under navnet *aktiveringsfunksjon*. Vet å sette uttrykket i likning 2 inn i likning 3 får vi følgende uttrykk, som representerer modellens prediksjon for target y:

$$y_{\text{pred}} = \frac{1}{1 + e^{-\sum_i w_i x_i + b}} \,. \tag{4}$$

1.2.1 Trening og tapsfunksjon

Før en modell kan gi oss gode prediksjoner for $\mathbf{x}$ må den tilpasses til dataene våre, og denne prosessen omtales som *læring* eller *trening* innen maskinlæring. Læringsprosessen går ut på at modellen tilpasses slik at måloppnåelsen øker på datasettet vi bruker. I maskinlæring omtales data som brukes til dette som *treningsdata*, og funksjonen som brukes til å beregne måloppnåelse som *tapsfunksjon*, fra engelske *loss function*. Denne funksjonen forteller oss hvor nærme modellens prediksjon er den riktige verdien, altså target, og vi bruker følgende notasjon

$$\mathcal{L}(y_{\text{pred}}, y) \tag{5}$$

for å angi en tapsfunksjon $\mathcal{L}$ som tar inn modellprediksjonene y_{pred} og targets y for å beregne tapet til modellen på det aktuelle datapunktet.

Neste spørsmål er hvilken tapsfunksjon vi skal velge. Dette er avhengig av oppgaven vi ønsker at modellen skal løse, og alternativene er mildt sagt mange. I vårt tilfelle ønsker vi å maksimere likelihood for riktig kategorisering (Survived=0 eller 1) av dataene, og et alternativ er følgende funksjon

$$p(y|\mathbf{x}) = y_{\text{pred}}^{y}(1 - y_{\text{pred}})^{1-y} \,. \tag{6}$$

Grunnen til at dette er en *likelihood* og ikke en sannsynlighet, er at vi *ikke* har lyst til å endre dataene slik at sannsynligheten for overlevelse endrer seg, men heller endre modellen slik at den estimerte sannsynligheten passer med den virkelige. **Oppgave:** Studer uttrykket i likning 6. Hva er verdien til dette hvis $y_{\text{pred}} = 1$ og $y = 1$? Hva med de andre kombinasjonene av de mulige verdiene til y_{pred} og y?

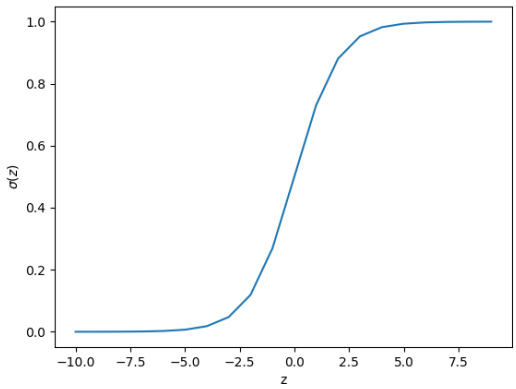


Figure 2: Sigmoid-funksjonen.

Uttrykket i likning 6 kan bli vanskelig å jobbe med på grunn av eksponentene. Siden vi kun ønsker å finne et optimum (minimum eller maksimum) i dette uttrykket, kan vi ta logaritmen, som er en monoton funksjon. Da har vi

$$\ln p(y|\mathbf{x}) = \ln\left(y_{\text{pred}}^{y}(1-y_{\text{pred}})^{1-y}\right) \tag{7}$$

$$= y\ln y_{\text{pred}} + (1-y)\ln(1-y_{\text{pred}})\,, \tag{8}$$

ofte omtalt som *log-likelihood*. Likelihood for å estimere riktig klasse bør *maksimeres*, så vi legger på et minustegn for å få en tapsfunksjon som skal *minimeres*. Da ender vi opp med såkalt *cross-entropy loss*:

$$\mathcal{L}(y_{\text{pred}}, y) = -\ln p(y|\mathbf{x}) \tag{9}$$

$$= -y\ln y_{\text{pred}} - (1-y)\ln(1-y_{\text{pred}}) \tag{10}$$

$$= -y\ln\sigma\left(\mathbf{w}\mathbf{x}+b\right) - (1-y)\ln\left(1-\sigma(\mathbf{w}\mathbf{x}+b)\right)\,. \tag{11}$$

Dette er tapsfunksjonen vi vil bruke for å tilpasse modellen til dataene.

Modellen vår er definert av parametrene $\mathbf{w}$ og b, som med kompakt notasjon kan skrives $\theta = (\mathbf{w}, b)$. Vi ønsker å finne parametrene som minimerer tapet, i gjennomsnitt over alle datapunktene. Dette kalles å *trene modellen*. Mer formelt vil vi velge parametrene θ som maksimerer log-likelihood for target-verdiene y for hvert datapunkt $\mathbf{x}$. Dette kalles *conditional maximum likelihood estimation*. For å trene modellen vår har vi altså følgende optimaliseringsproblem

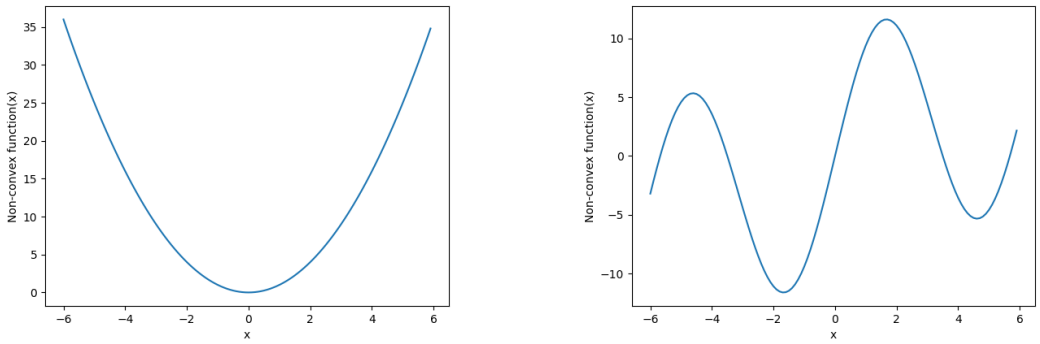
$$\hat{\theta} = \arg\min_{\theta} \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f(\mathbf{x}^{(i)};\theta), y^{(i)})\,. \tag{12}$$

For å løse dette optimaliseringsproblemet skal vi bruke *gradient descent*. Vi skal bruke den samme metoden senere i kurset når vi trener nevrale nettverk.

1.2.2 Gradient descent

Gradient descent er en metode for å finne minimumspunktet til en funksjon man ikke kjenner formen til. Strategien er å finne ut i hvilken retning i rommet definert av parametrene θ funksjonen minker brattest. Intuitivt kan vi se for oss at vi vandrer rundt på et fjell med bind for øynene, med mål om å finne det laveste punktet. Hvis man ikke kan se hvor det laveste punktet er, er beste strategi å føle seg frem til den bratteste nedoverbakken, og følge denne til man kommer ned fra fjellet.

En potensiell utfordring er at funksjonen kan ha både lokale og globale minima. Et lokalt minimum har en høyere funksjonsverdi enn et globalt minimum, men funksjonen har kun positive gradienter rundt minimumspunktet. I bind-for-øynene-analogien svarer dette til at man befinner seg i en fjellkjede,



(a) (b)

Figure 3: Eksempler på en (a) konveks, og en (b) ikke-konveks funksjon.

og kan gå seg vill mellom fjelltopper uten å vite at det finnes en dypere dal bak neste fjelltopp. Tapsfunksjonen for logistisk regresjon er konveks, og hvilket betyr at den kun har ett optimum. Figur 3a viser et eksempel på en konveks funksjon, mens figur 3b viser en ikke-konveks funksjon, med flere minima. Så lenge tapsfunksjonen vår er konveks har vi en garanti om at gradient descent som finner et minimum har funnet et globalt minimum. **Oppgave:** Hvilken retning bør vi bevege oss i? I hvilket rom?

Rommet vi beveger oss i er parameterrommet, altså rommet spent ut av θ-verdiene; hvis vi har fire parametre, er rommet fire-dimensjonalt. Vi bør bevege oss i den retningen der tapet minker. Det vi trenger er altså et mål på hvordan $\mathcal{L}$ endrer seg som funksjon av endring i θ. Vi trenger den deriverte av $\mathcal{L}$ med hensyn på θ, altså *gradienten* (dette er gradienten i "gradient descent"). Her lærer vi noe viktig, som du bør huske til senere: Siden gradient descent er veldig utbredt innen maskinlæring, også til trening av nevrale nettverk, forstår vi at *tapsfunksjonen må være deriverbar.*

I starten av treningen har alle parametrene en tilfeldig verdi, og vi oppdaterer verdiene iterativt, gjennom flere steg. Gradienten til tapsfunksjonen peker i retning av økende tap, og siden vi ønsker å minke tapet må vi bevege oss i motsatt retning. Dette representerer vi gjennom et minustegn, og regelen for parameteroppdateringen kan skrives som følger

$$\theta^{t+1} = \theta^t - \eta \frac{\partial}{\partial \theta} \mathcal{L}(f(\mathbf{x}; \theta), y)\,. \tag{13}$$

Den nye verdien θ^{t+1} etter steg t er den forrige verdien θ^t korrigert med gradienten til tapsfunksjonen. Parameteren η representerer hvor store korrigeringer som skal gjøres til parameterverdien, og omtales derfor som *læringsraten*: en høy verdi av η medfører store korrigeringer, mens en liten η medfører små endringer. Vi vet at modellen defineres av parametrene θ. Læringsraten η er også en parameter vi kan justere for at treningen skal gå best mulig, men denne er ikke en del av modellen. Det er en såkalt *hyperparameter*. Hyperparametre er parametre som definerer læringsprosessen, men ikke modellen. Det er viktig å forstå forskjellen på (modell)parametre og hyperparametre.

Vi trenger er uttrykk for den deriverte av tapsfunksjonen med hensyn på modellens parametre, altså hver w_i, og b. Det er en god øvelse å gjøre dette for hånd, og nyttige relasjoner er

$$\frac{\partial}{\partial x} \ln x = \frac{1}{x} \tag{14}$$

$$\frac{\partial}{\partial z} \sigma(z) = \sigma(z)\,(1 - \sigma(z)) \tag{15}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial x}\,. \tag{16}$$

Riktig uttrykk for de deriverte er

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{\partial}{\partial w_j}\left[-y \ln \sigma\left(\mathbf{w}\mathbf{x}+b\right) - (1-y)\ln\left(1-\sigma(\mathbf{w}\mathbf{x}+b)\right)\right] \tag{17}$$

$$= -y\frac{1}{\sigma(\cdot)}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\mathbf{x}+b) + (1-y)\frac{1}{1-\sigma(\cdot)}\frac{\partial}{\partial w_j}\left(1-\sigma(\mathbf{w}\mathbf{x}+b)\right) \tag{18}$$

$$= \frac{-y\left(1-\sigma(\cdot)\right)+(1-y)\sigma(\cdot)}{\sigma(\cdot)(1-\sigma(\cdot))}\frac{\partial}{\partial w_j}\sigma(\mathbf{w}\mathbf{x}+b) \tag{19}$$

$$= \frac{\sigma(\cdot)-y}{\sigma(\cdot)\left(1-\sigma(\cdot)\right)}\sigma(\cdot)\left(1-\sigma(\cdot)\right)\frac{\partial}{\partial w_j}(\mathbf{w}\mathbf{x}+b) \tag{20}$$

$$= \left(\sigma(\mathbf{w}\mathbf{x}+b)-y\right)x_j \tag{21}$$

$$= (y_{\text{pred}}-y)x_j \tag{22}$$

$$\frac{\partial \mathcal{L}}{\partial b} = \ldots \tag{23}$$

$$= \left(\sigma(\mathbf{w}\mathbf{x}+b)-y\right)\frac{\partial}{\partial b}(\mathbf{w}\mathbf{x}+b) \tag{24}$$

$$= \left(y_{\text{pred}}-y\right). \tag{25}$$

1.2.3 Trening

Vi har allerede kommet langt på vei for å gjøre maskinlæring: Vi har en modell, vi vet hvilke parametre som definerer den, vi har en tapsfunksjon, og vi har et uttrykk for å oppdatere parameterverdiene slik at tapet minker. Når vi har laget et program som utfører disse stegene, har vi laget en *læringsalgoritme*. En måte å strukturere et slikt program er gjennom følgende klasse og funksjonalitet (dere må skrive inn koden for funksjonene selv):

```
class LogisticRegression:
    def __init__(self, learning_rate=0.1, epochs=1000):
       self.learning_rate = learning_rate
       self.epochs = epochs
       self.weights, self.bias = None, None
       self.losses, self.train_accuracies = [], []

    def sigmoid_function(self, x):

    def _compute_loss(self, y, y_pred):

    def compute_gradients(self, x, y, y_pred):

    def update_parameters(self, grad_w, grad_b):

    def accuracy(true_values, predictions):
        return np.mean(true_values == predictions)
```

I tillegg trengs to viktige metoder: 1) `fit`, som tilpasser modellparametrene til treningsdataene. Her angir parameteren *epochs* hvor mange ganger gradient descent skal brukes på alle treningsdataene. 2) `predict`, som returnerer modellens prediksjon på data. Disse to navnene, `fit` og `predict`, er så standard i ulike maskinlæringsbiblioteker at man gjør lurt i å holde seg til dem og ikke dikte opp egne navn, selv når man skriver kode selv fra bunnen. Metodene kan for eksempel implementeres som følger

```
def fit(self, x, y):
   self.weights = np.zeros(x.shape[1]) #x.shape = datapunkter, features
   self.bias = 0

   # Gradient Descent
   for _ in range(self.epochs):
      lin_model = np.matmul(self.weights, x.transpose()) + self.bias
```

```
        y_pred = self._sigmoid(lin_model)
        grad_w, grad_b = self.compute_gradients(x, y, y_pred)
        self.update_parameters(grad_w, grad_b)

        loss = self._compute_loss(y, y_pred)
        pred_to_class = [1 if _y > 0.5 else 0 for _y in y_pred]
        self.train_accuracies.append(accuracy(y, pred_to_class))
        self.losses.append(loss)

def predict(self, x):
    lin_model = np.matmul(x, self.weights) + self.bias
    y_pred = self._sigmoid(lin_model)
    return [1 if _y > 0.5 else 0 for _y in y_pred]
```

Når koden for læringasalgoritmen er ferdig skrevet, er neste steg å ta i bruk dataene til å trene modellen. Dette bør vi *ikke* bruke hele datasettet til. Når vi er ferdige med å trene modellen har vi nemlig lyst til å undersøke hvor godt denne gjør det på data som ikke har blitt brukt til å tilpasse parametrene. Et sentralt mål innen maskinlæring er å lage modeller som klarer å *generalisere*, altså gjøre det godt på *nye* data. Derfor er det standard prosedyre å dele et datasett i minimum to deler, hvorav den ene delen er størst og utgjør *treningsdata*. Den andre, mindre delen representerer *testdataene*. Treningsdataene brukes til parametertilpasning, mens testdataene ikke røres før modellen er ferdig tilpasset, og brukes for å evaluere modellen før denne tas i bruk. For å dele datasettet i to kan metoden `train_test_split` fra biblioteket `scikit-learn` brukes.

Når læringsalgoritmen er på plass og dataene er splittet i en trenings- og en test-del, bør hele maskinlæringsprosedyren se ut omtrent som følger.

```
# Training
train_epochs = 30

# Initialize and train the model
log_reg = LogisticRegression_(learning_rate=0.01, epochs=train_epochs)
log_reg.fit(X_train, y_train)

# Make predictions
predictions = log_reg.predict(X_test)
```

1.2.4 Evaluering

Tenk på definisjonen av maskinlæring gitt i starten av kurset: For å vite at programmet faktisk blir bedre til å løse oppgaven gjennom erfaring (data), trenger vi en måte å måle ytelsen. Som et minimum bør vi lagre tapet (loss) og treffsikkerhet (accuracy) i løpet av treningen, og plotte disse etter treningen. Dette kan gjøres med følgende kode, ved hjelp av biblioteket `matplotlib`,

```
epoch_list = np.arange(0, train_epochs,1)
plt.plot(epoch_list, log_reg.losses, c='red', label="Loss")
plt.plot(epoch_list, log_reg.train_accuracies, c='blue', label="Accuracy")
plt.legend()
plt.show()
```

og et eksempel på resulterende plott fra en vellykket treningsprosedyre er vist i figur 4. På dette plottet ser vi at tapet er høyt i starten, og minker jevnt før det flater ut. Dette er bra; vi ønsker at tapet skal minke. Treffsikkerheten starter derimot lavt, og øker før den flater ut. Dette er også bra; vi ønsker økende treffsikkerhet. Til sammen gir disse to kurvene oss et bilde av en modell som lærer av dataene, og en treningsprosedyre som har foregått lenge nok. Hadde kurvene ikke flatet ut, hadde vi trengt flere epoker for at modellen skulle lære ferdig. I dette tilfellet kunne vi også stoppet treningen tidligere, etter rundt 20-25 epoker.

Selv om tap og treffsikkerhet kan fortelle oss om modellen lærer fra dataene, forteller de mildt sagt kun en liten del av historien, og gir oss ikke et godt bilde på hva modellen er god på eller hvilke typer

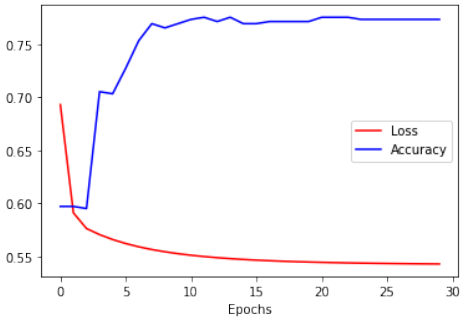


Figure 4: Eksempel på plott som viser tap (rød) og treffsikkerhet (blå) over epokene i en treningsprosedyre.

Table 4: Et utvalg metrikker vi kan beregne basert på verdiene i confusion matrix.

Betegnelse	**Uttrykk**	**Alternative navn**
True Positive Rate (TPR)	$\frac{TP}{P} = \frac{TP}{TP+FN}$	Sensitivitet, recall, sannsynlighet for deteksjon
True Negative Rate (TNR)	$\frac{TN}{N} = \frac{TN}{TN+FP}$	Spesifisitet
False Positive Rate (FPR)	$\frac{FP}{N} = \frac{FP}{FP+TN}$	Sannsynlighet for falsk alarm, (1 - spesifisitet)
Positive Predictive Value (PPV)	$\frac{TP}{TP+FP}$	Precision

datapunkter den typisk gjør feil på. De gir oss ikke en god anledning til å analysere modellens styrker og svakheter. Videre gir ikke tall som angir tap og treffsikkerhet intuitivt mening for mennesker. Vi bør derfor velge en eller flere *metrikker* for å angi modellens ytelse, og disse metrikkene bør gi mening for mennesker. I motsetning til tapsfunksjonen trenger ikke metrikken(e) vi velger å være deriverbar(e). **Oppgave:** Hva er flere grunner til at vi ikke bør velge samme tapsfunksjon og evalueringsmetrikk?

For klassifiseringsmodeller baseres mange metrikker på *confusion matrix*, hvis innhold angir hvor mange av datapunktene klassifiseres riktig (true) og feilaktig (false) i de ulike klassene. I vårt tilfelle har vi to klasser: 0 (negative), og 1 (positive), og vi kan sette opp en confusion matrix etter malen i figur 5. Her plasseres de sanne prediksjonene (true positive/negative), altså prediksjonene som samsvarer med target, på diagonalen, og de falske prediksjonene (false positive/negative) utenfor diagonalen i matrisen. Basert på verdiene i confusion matrix kan vi regne ut flere metrikker, for eksempel de angitt i tabell 4.

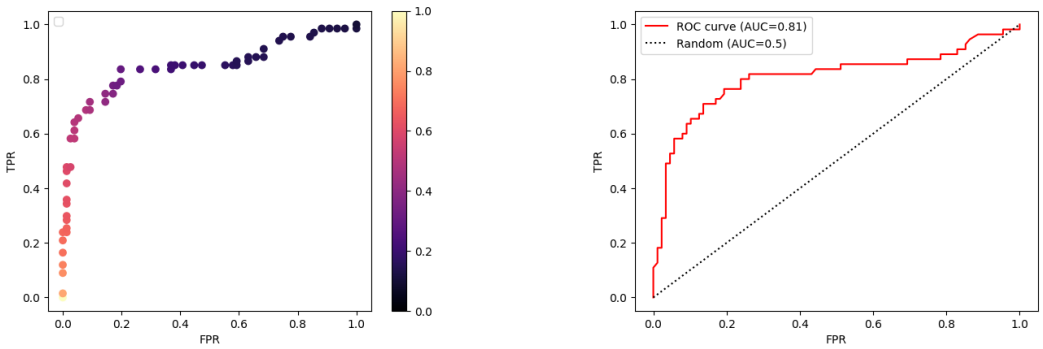
		Predicted	
	Total (P+N)	Negative (PN)	Positive (PP)
True	Negative (N)	True negative	False positive
	Positive (P)	False negative	True positive

Figure 5: Confusion matrix for binær klassifisering.

Oppgave: Du skal levere en maskinlæringsmodell til et sykehus som vil finne ut hvilke pasienter som lider av en dødelig sykdom. (I dette scenariet er det veldig alvorlig om modellen får en falsk negativ.) Hvilken metrikk bør modellen skåre høyt på? Senere skal du (du har mye å gjøre, ja) levere en modell til klimaaktivister som vil skyte raketter på tomme passasjerfly. (I dette scenariet er det alvorlig om modellen får en falsk positiv, fordi raketter er dyre.) Hvilken metrikk bør denne modellen skåre lavt på?

1.2.5 Klassifiseringsterskel

Modellen vår returnerer et tall i intervallet $(0, 1)$, og en enkel måte å gjøre dette om til en klassifisering er det vi gjorde i koden for `fit` og `predict`, nemlig følgende linje



(a) (b)

Figure 6: ROC-kurven, som viser FPR og TPR for ulike klassifiseringsterskler.

```
[1 if _y > 0.5 else 0 for _y in y_pred]
```

hvor vi tolker en prediksjon som `Survived=1` hvis `y_pred>0.5`, og ellers `Survived=0`. Her er tallet 0.5 klassifiseringsterskelen, og vi kunne i prinsippet valgt et annet tall. Hvis det var viktig for oss å være forsiktig, altså ikke predikere at en person vil overleve med mindre modellen er helt sikker, kunne vi valgt en terskel på for eksempel 0.7 i stedet. Hvilken terskel som bør velges er avhengig av problemet, og vi kan studere hvilket utslag ulike terskler har på ulike metrikker.

Det vanligste er å studere FPR og TPR som funksjon av klassifiseringsterskelen. Det gir oss Receiver Operating Characteristic (ROC)- kurven, se figur 6a. Her beregner vi både FPR og TPR for mange ulike klassifiseringsterskler fra 1 til 0. **Oppgave:** Ønsker vi høy eller lav verdi for henholdsvis FPR og TPR? Hva blir FPR og TPR for klassifiseringsterskel 0? Tilsvarende for 1?

Vi ønsker en høy TPR, altså sann positiv rate, mens vi ønsker at TPR, altså falsk positiv rate, holder seg lav. Hvis klassifiseringsterskelen er 1, vil nærmest ingen prediksjoner tolkes som positiv klasse, slik at både FPR og TPR er lave. Klassifiseringsterskel på 0 tilsier at nærmest alle prediksjoner tolkes som positiv klasse, hvilket vil gi høye verdier for både FPR og TPR. Derfor starter ROC-kurven i $(0, 0)$ og ender i $(1, 1)$. Det interessante er hva som skjer mellom disse punktene, særlig om TPR øker raskere enn FPR, og for hvilke klassifiseringsterskel som gir best trade-off mellom de to. **Oppgave:** Hvordan ser ROC-kurven ut for en modell som kaster mynt?

Uansett hvilken oppgave vi løser, ønsker vi en modell som har høy TPR og lav FPR: Des høyere TPR per FPR, des bedre er modellen, for alle terskler. Tilfeldig gjetning (myntkast) gir diagonalen $(0, 0) - (1, 1)$, den stiplede linjen i figur 6b. For å være bedre enn tilfeldig gjetning må kurven som representerer modellen derfor ligge over diagonalen, og arealet under kurven (forkortet AUC, fra area under curve) være større enn 0.5. Siden begge aksene går fra 0 til 1 er det største mulige arealet under kurven 1. Dette arealet, ROC AUC, representerer sannsynligheten for at modellen vil predikere en høyere verdi (nærmere 1 enn 0, for binær klassifisering) for et tilfeldig valgt positivt datapunkt enn for et tilfeldig valgt negativt datapunkt.

Utover ROC-kurven kan vi i prinsippet plotte hvilke som helst metrikker for varierende klassifiseringsterskler for å studere modellens oppførsel. En annen vanlig kurve å studere er den som viser *precision* og *recall*, se figur 7a. Presisjon angir andel korrekt predikert positive per predikert positive, altså

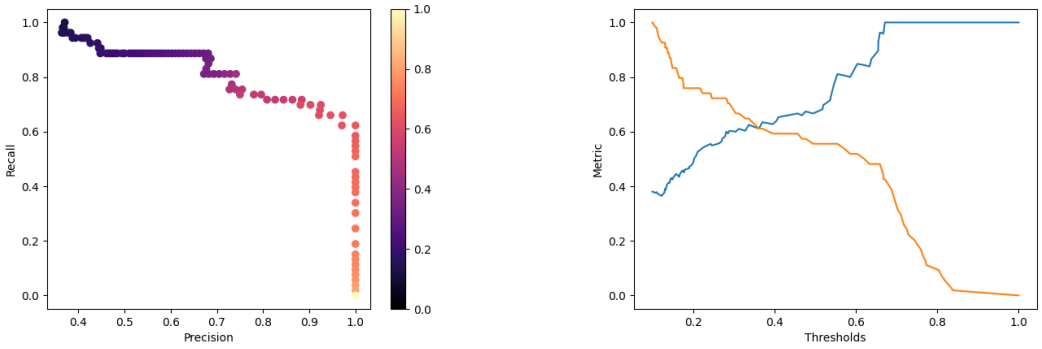
$$\text{Precision} = \frac{TP}{TP + FP}\,. \tag{26}$$

Høy presisjon svarer til lavt relativt antall falske alarmer, siden TP da dominerer over FP. Recall angir andel korrekt predikert positive per totalt positive, altså

$$\text{Recall} = \frac{TP}{TP + FN}\,. \tag{27}$$

Høy recall svarer til lavt antall missed cases, siden TP da dominerer over FN.

Når vi justerer klassifiseringsterskelen ser vi at det finnes en tradeoff mellom precision og recall; når den ene øker, minker den andre. Hva som er en god balanse mellom de to, er avhengig av hva modellen skal brukes til. Hvis falsk alarm er dyrt (for eksempel fører til en utrykning), er høy presisjon viktig. Hvis kostnaden med å bomme på en positiv instans er høy (for eksempel diagnostisering av alvorlige sykdommer), er høy recall viktig. Det kan være nyttig å for eksempel studere hvor precision- og recall-kurvene krysser hverandre for ulike klassifiseringsterskler, se figur 7b.



(a) (b)

Figure 7: (a) Precision-recall-kurven , og (b) henholdsvis precision- og recall-kurven, for ulike klassifiseringsterskler.

1.2.6 Kort om entropi og cross entropy loss

I informasjonsteori er entropi (også kalt Shannon entropy) en måte å kvantisere usikkerheten eller mengden informasjon tilgjengelig i en variabels mulige tilstander:

$$H(p) = -\sum_i p(x_i) \log p(x_i)). \tag{28}$$

I tilfellet med Titanic-dataene representerer $p(x_i)$ her sannsynligheten for `Survived`. Kryss-entropien (cross entropy) representerer da forskjellen mellom to ulike sannsynlighetsfordelinger p og q:

$$H(p, q) = -\sum_i p(x_i) \log q(x_i)) \,. \tag{29}$$

Sammenlikn dette med tapsfunksjonen i likning 9. Den ene sannsynlighetsfordelingen representerer labels i datasettet, mens den andre representerer modellens prediksjoner. Tapsfunksjonen forteller oss altså om *forskjellen* mellom de to fordelingene, og siden målet vårt er å lage en klassifiseringsmodell, ønsker vi at forskjellen mellom de to fordelingene er minst mulig. I tilfellet ikke-binær klassifisering, altså klassifisering til flere enn to klasser, generaliserer uttrykket i likning 29 til

$$\mathcal{L}(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i) \,, \tag{30}$$

hvor summen går over alle klassene i.

1.2.7 Bayes' teorem

Se på de to delvis overlappende mengdene A og B i figur 8. Sannsynligheten for å trekke et element fra overlappet $A \cap B$ er

$$P(A \cap B) = P(A|B)P(B) = P(B|A)P(A) \,. \tag{31}$$

Vi beholder den siste relasjonen, og omskriver den som følger

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \,. \tag{32}$$

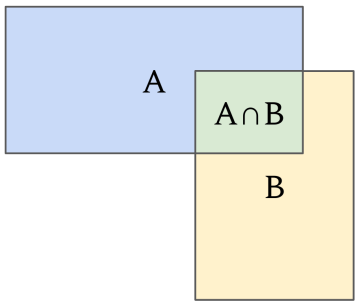


Figure 8: To delvis overlappende mengder A og B.

Dette er Bayes' teorem, og holder uansett sansynlighetstolkning. Når vi løser en klassifiseringsoppgave, som å beregne sannsynligheten for at en gitt passasjer overlever Titanic-ulykken, estimerer vi $p(C_k|x)$, altså sannsynligheten for klasse k (i vårt tilfelle kan k være 0 eller 1), betinget på observasjonen $\mathbf{x}$. For enklere notasjon dropper vi vektornotasjonen på $\mathbf{x}$, dvs vi skriver x uten fet skrift, i dette delkapitlet. Videre bruker vi Bayes' teorem for å få et uttrykk for denne sannsynligheten:

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)}\,. \tag{33}$$

De ulike leddene i denne likningen betyr følgende

- $p(C_k)$ er *a priori* sannsynligheten for klasse, altså fordelingen av klassene i treningsdataene. Denne omtales som *prior*, fordi det er sannsynligheten for klassetilhørighet vi kan estimere uten å vite noe om det aktuelle datapunktet (legg merke til at x ikke forekommer i dette leddet).
- $p(x)$ er fordelingen av selve dataene, uavhengig av klasse (legg merke til at C_k ikke forekommer i dette leddet). Dette er den faktiske sannsynlighetsfordeligen av alle feature-verdiene for alle dataene, og den har like mange dimensjoner som datasettet har features. Dette leddet omtales som *evidens*.
- $p(x|C_k)$ er den betingede sannsynlighetsfordelingen av dataene gitt klassen, og det er en *likelihood*. Det er altså en egen sannsynlighetsfordeling – med like mange dimensjoner som datasettet har features – for hver klasse, i vårt tilfelle to.

Når vi bruker maskinlæring, eller en hvilken som helst form for dataanalyse, til å estimere klassen til et datapunkt, er det nettopp $p(C_k|x)$ vi prøver å estimere. Gitt likning 33, vet vi at det finnes et *analytisk* uttrykk for $p(C_k|x)$. Hvis vi klarer å beregne alle leddene i likning 33 trenger vi derfor ikke å estimere sannsynligheten; vi kan i prinsippet bare beregne den nøyaktig. Hvorfor gjør vi ikke det?

La oss se på leddene i likning 33 for Titanic-dataene. Kan vi beregne alle leddene nøyaktig?

- $p(C_k)$ kan beregnes: Det er antallet henholdsvis Survived=0 og Survived=1 per totalt antall datapunter. Hvis 50% av passasjerende overlever, er dette leddet lik 0.5.
- $p(x)$ er problematisk å skrive ned, siden vi trenger informasjon om alle mulige verdier av alle featurene i datasettet, og det videre ikke er sikkert at vi vil ende opp med en funksjon med et analytisk uttrykk. Det kan derfor bli vanskelig å finne den faktiske verdien til sannsynligheten for et gitt datapunkt x. Men: siden dette leddet er uavhengig av klasse, vil det ha samme verdi for begge klassene. Så lenge vi er ute etter sannsynligheten for en klasse gitt data, kan vi derfor droppe det, da det bidrar like mye til begge klassene.

Basert på siste punkt over, kan vi droppe $p(x)$ i vårt tilfelle, og gjøre om sannsynligheten for klasse til følgende proporsjonalitet

$$p(C_k|x) \propto p(x|C_k)p(C_k)\,. \tag{34}$$

Nå gjenstår kun leddet som representerer likelihood for data gitt klasse:

- $p(x|C_k)$ er utfordrende å beregne så lenge features x er avhengige av hverandre, siden vi da må finne et uttrykk for en flerdimensjonal simultanfordeling. Dette er vanskelig (kanskje umulig) på samme måte som for evidensen $p(x)$.

En løsning som noen ganger fungerer i praksis, er å anta *uavhengige features*. For eksempel kan en passasjer være kvinne eller mann uavhengig av alder; verdien til feature "Sex" er *uavhengig* av feature "Age". Så lenge dette holder for alle features, kan vi skriv eom simultanfordelingen til et produkt av fordelinger over hver enkelt feature, og estimatet vårt forenkles til

$$p(C_k|x) \propto p(C_k) \prod_{i=1}^{n} p(x_i|C_k)\,. \tag{35}$$

Dette kalles *Naïv Bayes*, fordi det er naïvt å behandle features som uavhengige (selv om det i noen tilfeller kan stemme), og den tilsvarende sannsynligheten for klassetilhørighet er

$$\hat{y} = \underset{k\in\{1,\ldots,n\}}{\arg\max}\; p(C_k) \prod_{i=1}^{n} p(x_i|C_k)\,. \tag{36}$$

I forelesningene går vi igjennom et eksempel hvor vi estimerer sannsynlighet for overlevelse for to ulike tilfeller 0 og 1, og finner ut at denne er mindre for tilfelle 0 enn for tilfelle 1, altså

$$p(C_1) \prod_{i=1}^{n} p(x|C_1) > p(C_0) \prod_{i=1}^{n} p(x|C_0)\,. \tag{37}$$

1.2.8 Dimensjonsforbannelsen

Forelesningsnotater om dimensjonsforbannelsen del 1 mangler, og inntil videre er det slides fra forelesningene som gjelder. Se også avsnitt 3.1.4.

1.2.9 Trening på ubalanserte data

En masterstudent blir invitert til jobbintervju som ML-utvikler. Under intervjuet får kandidaten følgende spørsmål: "Vi har et datasett fra det norske helsevesenet, og skal lage en modell som klassifiserer friske og syke personer. Modellen vår får en treffsikkerhet (accuracy) på 99%, men den klarer nesten ikke å identifisere syke personer. Hva har skjedd?" **Oppgave:** Hva vil du svare på dette?

Dette vil typisk skje om den ene klassen (for eksempel den som representerer syke personer) inneholder mange færre datapunkter enn den andre. Førstnevne klasse er da *underrepresentert*, mens klassen med flere datapunkter er *overrepresentert*. Hvis forskjellen mellom antall datapunkter i de to klassene er veldig stor, vil modellen belønnes om den bare predikerer at alle datapunktene tilhører den overrepresenterte klassen. Dette kalles også *prior probability shift*, siden modellprediksjonen domineres av $p(C_k)$ i likning 33, og ikke av leddene som inneholder informasjon om feature-verdiene. Modellen vil slik oppnå høy treffsikkerhet, men likevel ikke være det minste nyttig. Mens en slik modell vil ha en høy treffsikkerhet og en høy ROC AUC, vil andre metrikker være lave. I dette tilfellet, der positiv klasse er underrepresentert, vil precision være forholdsvis lav, og recall vil være oppsiktsvekkende lav. Dette illustrerer at det er viktig å studere flere ulike metrikker for å forstå modellens svakheter. Videre forteller det oss at vi må gjøre noe mer for å få en modell til å bli nyttig når vi har skjevfordeling mellom klassene i (trenings-)datasettet.

Den enkleste tilpasningen vi kan gjøre er å endre klassifiseringsterskel i uttrykket

```
y_pred = [1 if _y > 0.5 else 0 for _y in y_pred]
```

fra 0.5 til en verdi som gir modellen høyere verdi på de andre metrikkene. Terskelen kan for eksempel bestemmes basert på precision-recall-plottet. Ved å senke terskelen kan vi øke andelen predicted positive, fordi alt over terskelen predikeres som 1. Å maksimere én metrikk går som oftest på bekostning av andre metrikker, og vi må huske at å justere denne terskelen *ikke* forbedrer modellen: det endrer kun hvordan vi forholder oss til modellens prediksjoner. Utover å endre terskelen kan vi også justere dataene eller tapsfuksjonen.

Resampling

Når vi gjør resampling for å gå fra et skjevfordelt til et jevnt fordelt datasett, har vi to muligheter.

- **Undersampling:** Her trekker vi like mange datapunkter fra den overrepresenterte klassen som vi har tilgjengelig i den underrepresenterte klassen. Da ender vi opp med et datasett bestående av like mange datapunkter fra hver klasse, men potensielt veldig få datapunkter totalt. Dette kan føre til at modellen som trenes på dataene undertilpasser (underfit).
- **Oversampling:** Her kopierer vi instanser fra den underrepresenterte klassen, inntil vi har like mange datapunkter fra den underrepresenterte som fra den overrepresenterte klassen. Da ender vi også opp med like mange datapunkter fra hver klasse, men potensielt mange duplikater fra den underrepresenterte klassen. Dette kan føre til at modellen som trenes på dataene overtilpasser (overfit).

Disse to teknikkene kan naturligvis kombineres, altså at man både undersampler den overrepresenterte klassen og oversampler fra den underrepresenterte klassen, slik at man til sammen har balanserte klasser i det resulterende datasettet. **Oppgave:** Vi har et datasett med 100 datapunkter fra den ene klassen, og 1000 fra den andre. Hvor mange datapunkter får vi totalt om vi gjør kun undersampling? Hvor mange får vi totalt om vi gjør kun oversampling, og hvor mange duplikater får vi?

Vektet tapsfunksjon

I stedet for å endre hvilke data modellen får tilgang til, kan vi fortelle modellen at en av klassene (ofte den underrepresenterte) er ekstra viktig. Dette gjør vi gjennom å straffe feilprediksjoner (gi høyere tap) på den viktigste klassen høyere relativt til de(n) andre klassen(e). Se igjen på tapsfuksjonen binary cross entropy fra likning 9:

$$\mathcal{L}(y_{\text{pred}}, y) = -y \ln y_{\text{pred}} - (1-y)\ln(1-y_{\text{pred}}) \tag{38}$$

Hvordan kan vi tilpasse denne slik at klassene vektes ulikt?

Hvis target $y = 0$ er første ledd null, og bidrar ikke til tapet. Kun andre ledd bidrar til tapet, som betyr at en vekting av andre ledd i likning 9 vil øke tapet for datapunkter med $y = 0$. Det samme gjelder for target $y = 1$: her er andre ledd null, og bidrar ikke til tapet. Vekting av det første leddet vil da føre til høyere loss for klassen med label $y = 1$. Den vektede tapsfunksjonen blir derfor

$$\mathcal{L}(y_{\text{pred}}, y) = -w_1 y \ln y_{\text{pred}} - w_0(1-y)\ln(1-y_{\text{pred}})\,, \tag{39}$$

hvor w_0 og w_1 representerer vekter for klasser 0 og 1. Hvis verdien på w_0 er betydelig større enn verdien på w_1 vil modellen prioritere å predikere riktig verdi for datapunkter med label $y = 0$, og vice versa.

1.3 Beslutningstrær

1.3.1 Noder

Beslutningstrær hjelper oss å ta beslutninger for datasett ved å splitte dataene i noder som til sammen utgjør en trestruktur, se figur 9a. *Rotnoden* (root node) er starten på treet, der data med alle features kommer inn. Under rotnoden finner vi *beslutningsnoder* (decision nodes). Rotnoden og beslutningsnodene har kriterier for å splitte dataene (splitting criteria). Beslutningsnoder er alle noder under rotnoden som har ett eller flere splitting criteria. Nederst i treet finner vi *løvnoder* (leaf nodes), som ikke splitter dataene. De inneholder predikert verdi for datainstansen som ble sendt gjennom treet. Beslutningstrær består av *trestumper* (tree stumps), se figur 9b. En trestump består av én rotnode og n løvnoder, for n mulige utfall. Vi holder oss til tilfellet $n = 2$, altså at to mulige utfall etter hver splitt. Hver beslutningsnode splitter dataene på én feature, gjennom et såkalt splitt-kriterium. Det samme treningsdatasettet og labels kan gi opphav til mange ulike beslutningstrær.

Som ellers i maskinlæring, bruker vi treningsdata for å bygge modellen, i dette tilfellet beslutningstreet. Treningsdataene brukes for å finne ut hvilke trestumper (og tilhørende beslutningskriterier) som bør settes sammen for å lage treet. Når vi bygger beslutningstrær ønsker vi alltid å velge det splitt-kriteriet som lar oss ta beslutningen tidligst mulig, altså reduserer usikkerheten mest mulig. Redusert usikkerhet er endringen i usikkerhet etter sammenliknet med før splitt. I hovedsak brukes følgende tre metrikker for å måle hvor mye et splitt-kriterium (feature og verdi) reduserer usikkerheten:

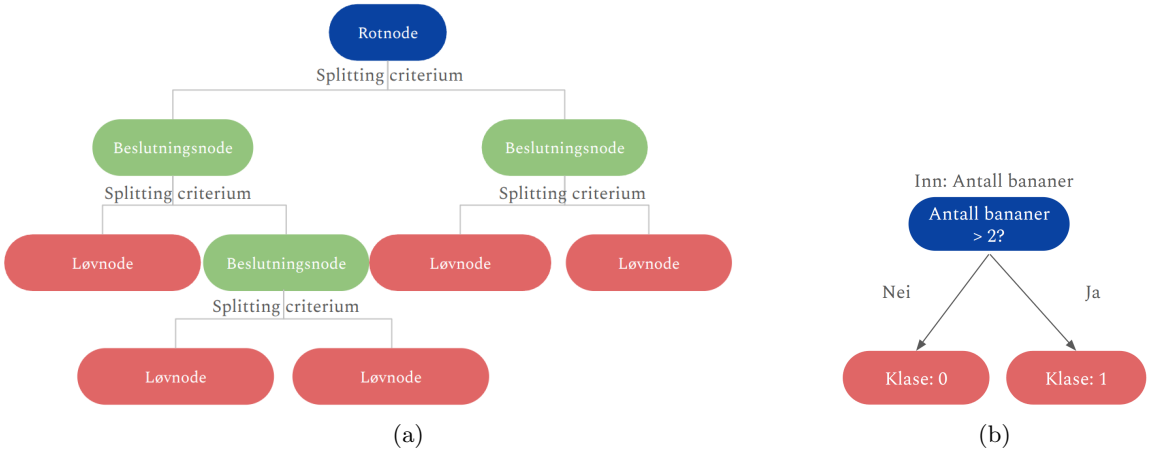


Figure 9: Skisser av (a) et beslutningstre, og (b) en stump.

- Log loss
- Gini impurity
- Entropi

Den første av disse, log loss, kjenner vi fra før, se likning 9 for binær klassifisering og likning 30 for multiklasse.

1.3.2 Gini Impurity

Gini-urenheten er et tall i $[0, 0.5]$ som angir sannsynligheten for at et nytt, tilfeldig datapunkt feilklassifiseres hvis det gis et tilfeldig label i henhold til klassedistribusjonen i datasettet. Gitt et datasett D bestående av datapunkter fra k klasser, med sannsynlighet p_i for at en instans tilhører klassen i ved en gitt node, er datasettets Gini-urenhet

$$\text{Gini}(D) = 1 - \sum_{i=1}^{k} p_i^2 \,. \tag{40}$$

Intuitivt: Du har en pose med kuler i ulike farger, hvor farge representerer klasse. Gini-urenheten måler hvor sannsynlig det er at du gjetter feil farge på en tilfeldig trukket kule, hvis du gjetter at kulens farge følger distribusjonen av farger i posen. Lav Gini-urenhet representerer scenariet der de fleste kulene har samme farge, slik at det er *lav* sannsynlighet for å gjette feil farge på en tilfeldig trukket kule. Datasettet regnes da å ha lav urenhet. Høy Gini-urenhet representerer motsatt scenario, der klassene er forholdsvis likt representert, og det er *høy* sannsynlighet for å gjette feil farge på en tilfeldig trukket kule. Datasettet regnes da å ha høy urenhet.

Hvis et dataset D splittes på feature f til to subsett D_1 og D_2 med henholdsvis n_1 og n_2 datapunkter, har vi

$$\text{Gini}_f(D) = \frac{n_1}{n}\text{Gini}(D_1) + \frac{n_2}{n}\text{Gini}(D_2) \,. \tag{41}$$

Table 5: Eksempler på Gini-urenhet for ulike splits.

	Antall		**Sannsynlighet**		**Gini**
	n_1	n_2	p_1	p_2	$1 - p_1^2 - p_2^2$
A	0	10	0	1	$1 - 0^2 - 1^2 = 0$
B	3	7	0.3	0.7	$1 - 0.3^2 - 0.7^2 = 0.42$
C	5	5	0.5	0.5	$1 - 0.5^2 - 0.5^2 = 0.5$

Oppgave: Se på tabell 5. Hvilken splitt bør vi velge, og hvorfor?

1.3.3 Entropi

Gitt en sannsynlighetsfordeling over k klasser, er sannsynligheten for hver klasse p_i . Entropien til fordelingen er

$$\mathrm{I}(p_1, \ldots, p_k) = -\sum_{i=1}^{k} p_i log_2(p_i)\,. \tag{42}$$

Oppgave: Hva er entropien til fordelingen $(0.5, 0.5)$, og fordelingen $(0.01, 0.99)$?

Den maksimale entropien til en binær fordeling er 1, som svarer til lik sannsynlighet per klasse. Den minimale entropien er 0, som svarer til at alle datapunktene tilhører samme klasse.

Før vi splitter på en feature har vi et datasett D med p positive og n negative instanser. Dette svarer til en binær distribusjon der positiv klasse har sannsynlighet $\frac{p}{p+n}$, og negativ klasse har sannsynlighet $\frac{n}{p+n}$, og entropien er

$$\mathrm{I}_{\text{før}} = \mathrm{I}\left(\frac{p}{p+n}, \frac{n}{p+n}\right)\,. \tag{43}$$

Etter at treet splitter på en feature, har vi to nye datasett, D_1 og D_2 (ett på hver side av splitten). Den forventede entropien etter denne splitten er generelt

$$\mathbb{E}[\mathrm{I}_{\text{etter}}] = \sum_{i=1}^{s} \frac{p_i + n_i}{p+n} \mathrm{I}\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)\,. \tag{44}$$

hvor s angir antall splitt, i vårt tilfelle $s = 2$, slik at $i = 1, 2$. Vi er interessert i forskjellen i entropi, altså entropien før og etter en valgt splitt:

$$\Delta\mathrm{I} = \mathrm{I}_{\text{før}} - \mathbb{E}[\mathrm{I}_{\text{etter}}] = \mathrm{I}\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - \sum_{i=1}^{s} \frac{p_i + n_i}{p+n} \mathrm{I}\left(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i}\right)\,, \tag{45}$$

og vi bør velge den splitten som gir størst endring i entropi. **Oppgave:** Vi har en beslutningsnode bestående av $p = 9$ positive og $n = 5$ negative datapunkter. Vis at $I_{\text{før}} = 0.94$. Anta deretter at vi splitter datasettet slik at vi får to nye datasett med henholdsvis $(p_1 = 4, n_1 = 5)$ og $(p_2 = 5, n_2 = 0)$. Regn ut endringen i entropi (svaret står rett under, men prøv selv).

$$\mathbb{E}[\mathrm{I}_{\text{etter}}] = \frac{4+5}{9+5}\mathrm{I}\left(\frac{4}{9}, \frac{5}{9}\right) + \frac{5}{9+5}\mathrm{I}\left(\frac{5}{5}, \frac{0}{5}\right) = -\frac{9}{14}\left(\frac{4}{9}\log_2\frac{4}{9} + \frac{5}{9}\log_2\frac{5}{9}\right) + 0 \approx 0.64\,. \tag{46}$$

1.3.4 Bygge beslutningstrær

I dette kurset trenger dere ikke bygge beslutningstrær selv, men dere bør kunne angi pseudokode for å bygge et beslutningstre. For å lage beslutningstrær bruker vi som oftest biblioteket `sklearn`, med den innebygde `DecisionTreeClassifier`. Denne lar oss velge mellom de splitt-kriteriene Gini impurity, entropy og log loss.

Oppgave: Plott Gini impurity og entropi i intervallet $[0, 1]$. Er kurvene like?

Gini impurity og entropi oppfører seg likt, og vil i hovedsak gi opphav til samme feature splits. Hovedforskjellen mellom de to ligger i at beregning av entropi krever en ekstra logaritme, som koster mer å beregne. Gini impurity er derfor mindre beregningstungt, altså raskere å regne ut. Denne er også default i **sklearn.DecisionTreeClassifier**.

Oppgave: Sammenlikn uttrykkene for log loss, entropi og Gini-urenhet. Likner noen av disse? Hvilken går raskest å evaluere?

Utover valg av beslutningskriterier, må vi besvare to ytterligere spørsmål:

- Når skal vi slutte å splitte, selv om nederste node har instanser fra begge klassene?
- Hva gjør vi med eventuelle løvnoder som har instanser fra begge klassene?

Tre tilfeller kan oppstå der vi må slutte å splitte: Når

1. det ikke finnes flere features å splitte på. Vi kan havne i en situasjon der vi har splittet på alle tilgjengelige features, men fremdeles ikke har klart å lage løvnoder som tilordner alle treningsdatapunktene til riktig klasse. Hvis vi ikke har flere features å splitte på, kan vi ikke lage en beslutningsnode: da er inneværende node nødvendigvis en løvnode, selv om den inneholder datapunkter fra flere klasser,
2. det ikke finnes flere datapunkter å teste. Vi kan havne i en situasjon der alle kombinasjoner av features som er *tilgjengelig i dataene* har blitt testet, uten at alle tenkelige kombinasjoner av features er testet,
3. treet har nådd en predefinert maksimal dybde. Dette er en hyperparameter som velges før man begynner å bygge treet.

Etter at treet er bygget vil vi sannsynligvis ha løvnoder som inneholder treningsdatapunkter fra begge klasser. For å bestemme hvilken beslutning en slik node kan ta har vi flere muligheter, hvorav de vanligste er å returnere

1. den dominante klassen i noden, altså label tilsvarende den dominante klassen fra treningsdataene i løvnoden,
2. et tilfeldig trukket label fra treningsdataene, altså *a priori*-sannsynligheten.

Vi kan nå sette sammen alt vi har vært gjennom til en algoritme som rekursivt bygger et beslutningstre, se pseudokode under.

```
def build_decision_tree(data, features, depth):
    if (all data in same class):
        return class label
    elif (no features left to test) or (depth >= max_depth):
        return 1 if p/(p+n) > n/(p+n) else 0
    elif (no data left to test):
        return (dominant class in parent node)
    else:
        choose best_feature f for split
        partition data based on f
        for each data partition:
            build_decision_tree(data partition, features - best_feature, depth+1)
```

1.4 Regresjon

1.4.1 Data og tapsfunksjon

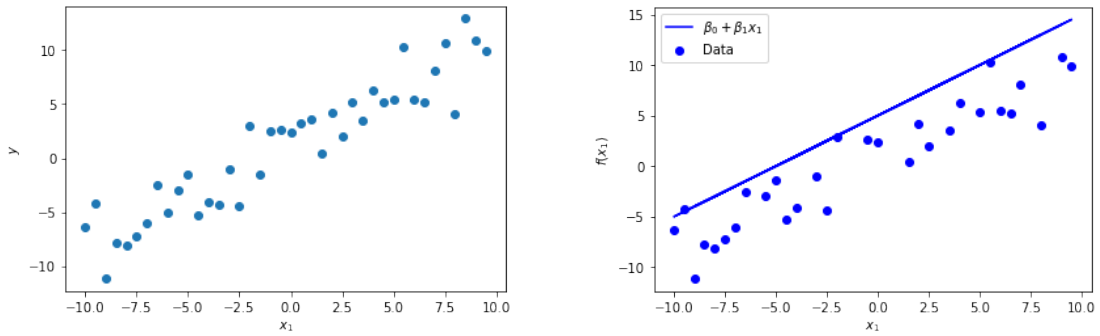
Se på datasettet i tabell 6, og tilhørende figur 10a. Igjen representerer y targets ("riktig svar"), så vi holder oss i regimet til veiledet læring. Men da vi gjorde klassifisering hadde vi targets som fordelte seg i predefinerte kategorier (to kategorier i det binære tilfellet). Det nye dataene våre hører ikke til kategorier, men har kontinuerlige verdier. Disse er kvantitative, og vi kan ikke gjøre klassifisering. Prediksjon til kontinuerlige verdier kalles *regresjon*. Eksempler på regresjonsoppgaver inkluderer å predikere skåren til en film, estimere formuesverdien til en bolig, forutse sykefraværet til en ansatt, osv. Generelt: estimering av en (eller flere) kontinuerlig(e) verdi(er). Vi starter igjen med det enkleste tilfellet, og den enkleste regresjonsmodellen er en lineær modell. Likningen for en lineær regresjonsmodell er

$$f(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n \,. \tag{47}$$

Modellen har en parameter β_i per dataegenskap x_i (kolonne i datasettet), og en parameter β_0 kalt bias. Til sammen har vi $(n+1)$ parametre for n features. Vi kan gjenbruke mesteparten av koden fra klassifiseringen for å trene denne modellen, altså tilpasse parametrene basert på dataene. Vi må igjen bruke gradient descent for å optimalisere parametrene.

Oppgave: Hva kan vi ikke gjenbruke fra klassifiseringstilfellet?

Vi trenger en tapsfunksjon som guider læringsalgoritmens justering av modellparametrene, altså et kvantitativt mål på hvor godt modellprediksjonen passer med target per datainstans. Regresjonsmod-



(a) (b)

Figure 10: (a) Datasett til lineær regresjon, og (b) en lineær regresjonsmodell med høy bias.

ellen vår predikerer verdier for y; ikke sannsynligheter. Den vanligste tapsfunksjonen innen regresjon er *mean squared error*:

$$\text{MSE} = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - f(\mathbf{x}^{(i)})\right)^2 . \tag{48}$$

Her er $y^{(i)}$ er target for datapunkt $\mathbf{x}^{(i)}$, f er modellen, og summen (gjennomsnittet) går over alle N instansene (radene) i datasettet. Faktoren $\frac{1}{2}$ er kun for convenience, og det finnes definisjoner av MSE der den ikke er med. MSE er liten hvis modellens prediksjon $f(\mathbf{x})$ er nærme den sanne verdien y, og stor hvis de to er ulike.

Hvis vi har et datasett med én feature x_1, har den lineære regresjonsmodellen to parametre, og kan skrives

$$f(x) = \beta_0 + \beta_1 x_1 . \tag{49}$$

Denne likningen er på samme form som likning 2, men med andre parameternavn, valgt for å skille dette tilfellet tydelig fra logistisk regresjon. Bias-parametren β_0 i likning 49 angir hvor regresjonslinjen krysser y-aksen, og er uavhengig av verdien til x_1. β_1 angir stigningstallet. Den deriverte av tapsfunksjonen med hensyn på de to parametrene er

$$\frac{\partial \mathcal{L}}{\partial \beta_0} = \frac{1}{2N}\sum_{i=1}^{N}\frac{\partial}{\partial \beta_0}\left(y^{(i)} - \beta_1 x^{(i)} - \beta_0\right)^2 \tag{50}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\left(\beta_1 x^{(i)} + \beta_0 - y^{(i)}\right) \tag{51}$$

$$\frac{\partial \mathcal{L}}{\partial \beta_1} = \frac{1}{N}\sum_{i=1}^{N} x^{(i)}\left(\beta_1 x^{(i)} + \beta_0 - y^{(i)}\right) . \tag{52}$$

Her ser vi nytten av faktoren $\frac{1}{2}$: den oppheves av potensen under derivasjonen. Vi kan kontrollere at de deriverte gir mening gjennom kontrollspørsmål.

Oppgave: Modellen oppfører seg som vist i figur 10b, altså den overestimerer konsekvent for alle verdier av x_1. Har modellen for lav eller for høy verdi av β_0? Er verdien til $\frac{\partial \mathcal{L}}{\partial \beta_0}$ større eller mindre enn 0?

En modell med for høy bias har $\frac{\partial \mathcal{L}}{\partial \beta_0} > 0$, som vil si at tapet øker når β_0 øker.

Oppgave: Hilken tilsvarende sjekk kan vi gjøre for β_1?

Fra gradienten til tapsfunksjonen får vi følgende oppdateringsregel til parametrene i regresjonsmodellen vår

$$\beta_0 \leftarrow \beta_0 - \eta\frac{1}{N}\sum_{i=1}^{N}(f(x^{(i)}) - y^{(i)}) \qquad \beta_1 \leftarrow \beta_1 - \eta\frac{1}{N}\sum_{i=1}^{N} x^{(i)}(f(x^{(i)}) - y^{(i)}) . \tag{53}$$

Mye av koden fra klassifiseringsoppgaven kan gjenbrukes for å tilpasse modellen i likning 49 til et gitt datasett.

Table 6: Datasett for regresjon.

Datapunkt	x_1	x_2
1	-10.0	-6.359354964838024
2	-9.5	-4.206219476164554
3	-9.0	-11.127060105339346
4	-8.5	-7.771131386305228
5	-8.0	-8.124549343247278
6	-7.5	-7.182177496316401
7	-7.0	-6.037172478551233
8	-6.5	-2.504149653937351
9	-6.0	-5.026172414497464
10	-5.5	-2.950079094157873
11	-5.0	-1.4507431897248368
12	-4.5	-5.336622551581822
13	-4.0	-4.059330972662908
14	-3.5	-4.327888761967305
15	-3.0	-1.0623486580711823
16	-2.5	-4.409261395640257
17	-2.0	2.9475603563217945
18	-1.5	-1.5395408141822513
19	-1.0	2.4622402587375745
20	-0.5	2.642472707669069
21	0.0	2.31891903693072
22	0.5	3.2425842723044016
23	1.0	3.537834860795493
24	1.5	0.3798562973180845
25	2.0	4.140802310836296
26	2.5	1.9614206593287309
27	3.0	5.162903327838864
28	3.5	3.5197438184133043
29	4.0	6.243395322509395
30	4.5	5.141524420933673
31	5.0	5.370176682456397
32	5.5	10.254373431713347
33	6.0	5.43525345790149
34	6.5	5.159899203707566
35	7.0	8.114723631612268
36	7.5	10.611562465835878
37	8.0	4.08003418246534
38	8.5	12.898017911956575
39	9.0	10.838245427992971
40	9.5	9.886185758809033

1.4.2 Bias og varians

Bias-leddet β_0 er funksjonsverdien i $x_1 = 0$. Dette leddet forflytter alle modellens prediksjoner med en konstant verdi. Tenk på det som en "baseline"-funksjonsverdi uten featureverdiene, som justeres til modellens prediksjon basert på featureverdiene.

Når vi snakker om bias til en (hvilken som helst) modell (ikke bare lineær regresjon), snakker vi om forskjellen mellom den sanne verdien vi prøver å estimere, og forventningsverdiene til estimatet verdiene, altså mellom targets y og prediksjoner $\hat{y}$. Vi har

$$\text{Bias}(y, \hat{y}) = \mathbb{E}[\hat{y} - y] = \mathbb{E}[\hat{y}] - y\,, \tag{54}$$

hvor siste likhet følger av at den faktiske verdien til y er målbar, mens $\hat{y}$ er en estimator, beregnet på et tilfeldig utvalg datapunkter. Dette utvalget er trukket fra en sannsynlighetsfordeling, og estimatoren får ulike verdier for ulike tilfeldige trekninger som lager treningsdataene. Denne iboende tilfeldigheten gjør at $\hat{y}$ er en tilfeldig variabel, med en sannsynlighetsfordeling. Derfor gir det mening å snakke om forventningsverdien til $\hat{y}$.

Oppgave: Gjennomsnittshøyden til norske kvinner er 167 cm. Vi lager en modell som estimerer høyde basert på andre egenskaper, og denne predikerer *i gjennomsnitt* at kvinner er 165 cm høye. Hvor stor bias har denne modellen?

Et annet nyttig statistisk mål er *varians*, som forteller oss om spredningen prediksjonene har fra gjennomsnittsverdien. Konkret er variansen til en tilfeldig variabel lik forventningsverdien til kvadrert avvik fra variabelens forventede verdi $\mathbb{E}[\hat{y}]$:

$$\text{Var}[\hat{y}] = \mathbb{E}\left[(\hat{y} - \mathbb{E}[\hat{y}])^2\right] \tag{55}$$

Se også figur 11. Merk at denne likningen kun inneholder predikerte verdier, og derfor fordeller oss om variansen til prediksjonene. Dette er nyttig, da

- høy varians ofte indikerer at modellen er i overkant sensitiv til variasjoner i data, hvilket tyder på overtiplasning (overfit),
- lav varians ofte indikerer at modellen predikerer for nært gjennomsnittsprediksjonen, og ikke gjør tilstrekkelig nytte av informasjonen i features, hvilket tyder på undertilpasning (underfit).

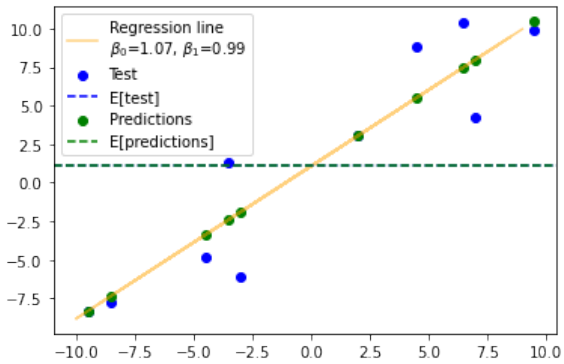


Figure 11: En lineær regresjonsmodell med testdata (blå punkter), prediksjoner (grønne punkter), og gjennomsnittsverdi for y og prediksjoner (blå/grønn stiplet linje).

For én og samme modell finnes det en avveining mellom bias og varians, kjent som *bias variance tradeoff*. Vi kan utlede denne gjennom å bruke følgende nyttige relasjoner til å skrive om MSE-tapsfunksjonen,

$$\mathbb{E}[c] = c \tag{56}$$
$$\mathbb{E}[cX] = c\mathbb{E}[X] \tag{57}$$
$$\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y] \tag{58}$$
$$\mathbb{E}[XY] \overset{!}{=} \mathbb{E}[X]\mathbb{E}[Y] \quad \textit{for statistisk uavhengige } X \textit{ og } Y. \tag{59}$$

Vi dropper her konstanten $\frac{1}{2}$ for en ryddigere utledning, og skriver

$$\begin{aligned}
\text{MSE}[y, \hat{y}] &= \mathbb{E}\left[(y - \hat{y})^2\right] && (60)\\
&= \mathbb{E}\left[(y - \mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}] - \hat{y})^2\right] && (61)\\
&= \mathbb{E}\left[y^2 - 2y\mathbb{E}[\hat{y}] + 2y\mathbb{E}[\hat{y}] - 2y\hat{y} + \mathbb{E}[\hat{y}]^2 - 2\mathbb{E}[\hat{y}]^2 + 2\mathbb{E}[\hat{y}]\hat{y} + \mathbb{E}[\hat{y}]^2 - 2\mathbb{E}[\hat{y}]\hat{y} + \hat{y}^2\right] && (62)\\
&= \mathbb{E}\left[(y - \mathbb{E}[\hat{y}])^2 + (\mathbb{E}[\hat{y}] - \hat{y})^2 + 2\left(y\mathbb{E}[\hat{y}] - y\hat{y} - \mathbb{E}[\hat{y}]^2 + \mathbb{E}[\hat{y}]\hat{y}\right)\right]. && (63)
\end{aligned}$$

Nå kan vi ta forventningsverdien separat for de tre leddene. For det første leddet har vi

$$\begin{aligned}
\mathbb{E}\left[(y - \mathbb{E}[\hat{y}])^2\right] &= \mathbb{E}\left[y^2 - 2y\mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}]^2\right] && (64)\\
&= y^2 - 2y\mathbb{E}[\hat{y}] + \mathbb{E}[\hat{y}]^2 && (65)\\
&= (y - \mathbb{E}[\hat{y}])^2 && (66)\\
&= \text{Bias}(y, \hat{y})^2, && (67)
\end{aligned}$$

og andre ledd kan vi sammenlikne med likning 55 for å identifisere

$$\mathbb{E}\left[(\mathbb{E}[\hat{y}] - \hat{y})^2\right] = \text{Var}(\hat{y}). \tag{68}$$

Siste ledd blir

$$2\mathbb{E}\left[y\mathbb{E}[\hat{y}] - y\hat{y} - \mathbb{E}[\hat{y}]^2 + \mathbb{E}[\hat{y}]\hat{y}\right] = 2\left(y\mathbb{E}[\hat{y}] - y\mathbb{E}[\hat{y}] - y\mathbb{E}[\hat{y}]^2 + \mathbb{E}[\hat{y}]^2\right) = 0 \tag{69}$$

Vi samler leddene og innser at vi kan skrive tapet som

$$\text{MSE}[y, \hat{y}] = \text{Bias}(y, \hat{y})^2 + \text{Var}(\hat{y}). \tag{70}$$

Denne relasjonen forteller oss at en modells tap består av en komponent som skyldes bias, og en som skyldes varians. I noen definisjoner forekommer også et ekstra ledd som representerere irredusibel feil grunnet støy, men vi vil ikke analysere denne. Det viktige for oss er å vite at relasjonen ofte omtales som *the bias variance tradeoff*, og hva den beskriver.

Noen ganger snakker vi også om the bias variance *dilemma*, fordi forsøket på å minke disse to kildene til prediksjonsfeil samtidig fører til en konflikt: Husk at vi alltid ønsker at en modell tilpasset på treningsdata skal generalisere godt til testdata. Høy bias representerer en feil generalisering på treningsdataene, som overskygger mer subtile, men faktisk relevante relasjoner mellom features og targets. Høy varians representerer at modellen har for høy sensitivitet til variasjoner i treningsdataene (inkludert støy), altså har overmodellert sammenhenger i stedet for å generalisere.

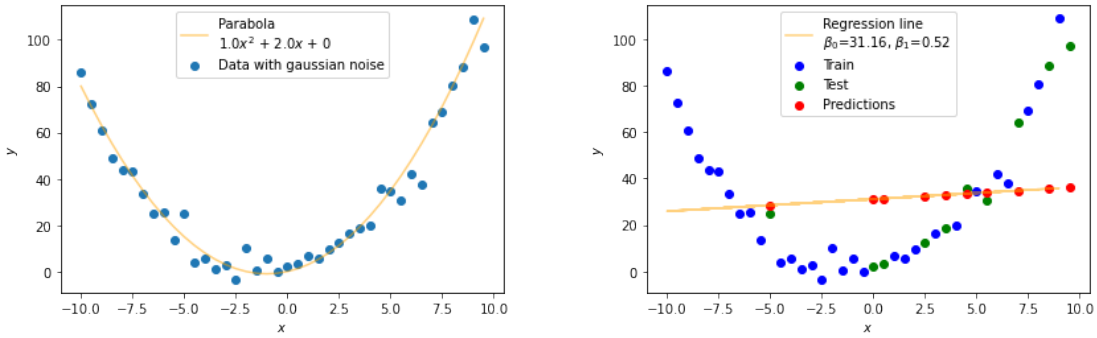
1.4.3 Feature engineering

Se på datasettet vist i figur 12a. Den oransje linjen representerer den underliggende fordelingen, og de blå punktene representerer datapunkter generert fra denne fordelingen, med tilfeldig støy trukket fra en gaussisk fordeling (med forventningsverdi $\mu = 0$ og standardavvik $\sigma = 0.5$) lagt til.

Oppgave: Hvor godt vil lineær regresjon fungere for å lage en modell som predikerer y for nye datapunkter x? Hvorfor? Prøv å formulere svaret ved hjelp av *antakelsen* som ligger i lineær regresjon.

Figur 12b viser en lineær modell tilpasset til disse dataene, og denne bekrefter det vi nok forventet: Lineær regresjon fungerer dårlig for å modellere en parabel. For å løse dette har vi flere muligheter, og i det aktuelle tilfellet kjenner vi *funksjonsformen* til fordelingen som genererte dataene. At vi kjenner funksjonen som ble brukt til å generere dataene vil så godt som aldri være tilfelle, men bare ved å plotte dataene kunne vi se at de fordeler seg som et annengrads polynom. Basert på er den enkleste løsningen å lage en ny feature, hvor vi transformerer dataene våre polynomisk. Dette kalles *feature engineering*.

Siden vi ser at dataene ligger på en parabel, bør vi velge transformasjonen $x \to x^2$. Vi kan gjøre dette for hånd, eller ved hjelp av `sklearn` sin innebygde `PolynomialFeatures`, som lar oss angi graden til polynomet vi ønsker å generere. Da lager og anvender vi transformasjonen på treningsdataene, og



(a) (b)

Figure 12: (a) Datasett (blå punkter) generert av en polynomisk likning av grad to (oransje linje) med gaussisk støy, og (b) en lineær regresjonsmodell tilpasset dette datasettet.

anvender den på testdataene. Det er viktig at vi ikke lager transformasjonen på testdataene, siden hele analysen skal være uavhengig av testdataene, og disse kun skal brukes for endelig testing. Etter transformasjonen har vi generert en ny feature, og der vi før hadde én feature x, har vi nå to features $\{x_1, x_2\} = \{x, x^2\}$, og har dermed økt dimensjonaliteten til dataene fra én til to. Vi skal derfor gjøre en ny lineær regresjon, denne gangen med to features, og modellen som skal tilpasses er

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 \,. \tag{71}$$

Merk at vi fremdeles gjør lineær regresjon, men nå med en generert feature. Merk også at feature engineering kan gjøres i alle tilfeller; ikke kun for veiledet læring, og ikke kun for regresjon.

Generelt er feature engineering alle operasjoner vi utfører på datasettet vi bruker til maskinlæring, og inkluderer:

- **Feature selection**, altså utvelgelse av features, som da vi valgte å ikke ta med "Name" i klassifiseringsoppgaven på Titanic-dataene.
- **Feature preprocessing**, altså preprosessering av features, som da vi skalerte "Age" til intervallet $(0, 1)$ for Titanic-dataene.
- **Feature extraction**, altså utvinning av features, som da vi nettopp laget en feature x^2 fra x. Det er også mulig å kombinere flere eksisterende features til nye.

Det finnes mange metoder og triks for feature engineering, og vi kommer mildt sagt ikke til å dekke alle i dette kurset. Dere kan ta i bruk metoder etter eget ønske og fantasi når dere løser øvingsoppgavene, gitt at *dere forstår og kan forklare hva dere har gjort og hvorfor.*

For én variabel x kan vi skrive et polynom av grad M som

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \cdots + \beta_M x^M \,. \tag{72}$$

og figur 13 viser data generert fra et polynom av ukjent grad.

Oppgave: Er det mulig å se fra dataene i figuren hvilken grad vi bør generere features til?

Figurene 14a, 14b og 14c viser lineær regresjon til dataene i figur 13, med feature transformasjoner til ulike grader. Her ser vi at en mer kompleks modell (høyere grads polynom) ikke nødvendigvis gir lavere tap (MSE), men at både for lav og for høy kompleksitet i modellen er problematiske.

Oppgave: Hvilken modell i figurene har for høy varians? Hvilken har lavest bias?

Oppgave: Er graden på polynomet vi transformerer features til en modellparameter, en hyperparameter eller en feature?

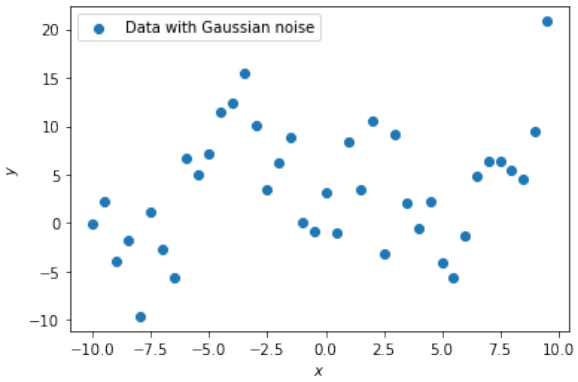
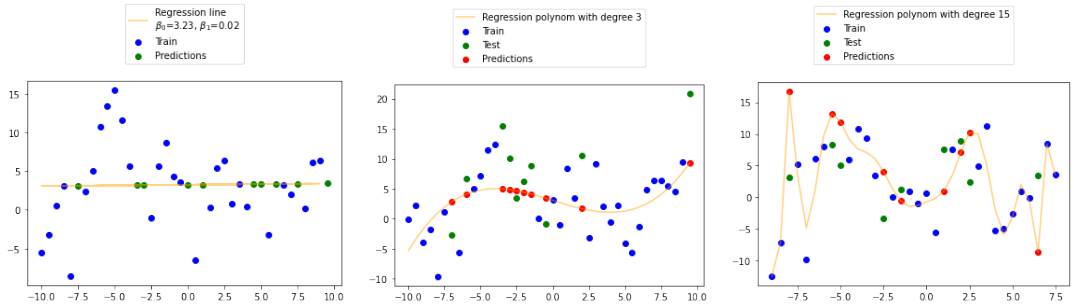


Figure 13: Datasett generert fra polynom av ukjent grad, med gaussisk støy



(a) (b) (c)

Figure 14: (a) Lineær regresjon og regresjon med genererte features til grad (b) 3 og (c) 15, til datasettet representert av de blå punktene.

1.4.4 Trening, testing og validering

Overtilpasning skjer når modellen har kapasitet til å tilpasse seg så godt til treningsdataene at det går utover generaliseringsevnen på testdataene. Det er viktig å kunne detektere overtilpasning, slik at vi kan justere hyperparametrene underveis i treningen. Husk: Testdataene skal ikke brukes til å justere treningen; heller ikke hyperparametrene. Likevel kan vi ikke bruke treningsdataene til å detektere overtilpasning, siden treningsprosedyren har som mål at modellen skal tilpasses best mulig til nettopp disse. Siden hverken trenings- eller testdataene kan brukes til å tilpasse hyperparametrene, trenger vi ytterligere data til dette formålet. Det vanligste er å dele det opprinnelige datasettet i tre deler:

- **Treningsdata** brukes for å tilpasse modellparametrene, altså under trening.
- **Testdata** brukes ikke til å gjøre noen som helst tilpasning av modellen, parametre, hyperparametre eller treningsprosedyren: De tas i bruk for å rapportere ytelsen til den endelige modellen.
- **Valideringsdata** brukes for å monitorere modellen under trening, og på bakgrunn av dette til å justere hyperparametrene.

Metoden `train_test_split` fra `sklearn.model_selection` kan brukes til å splitte det opprinnelige datasettet i to, to ganger. Til sammen ender vi da opp med tre uavhengige datasett, hvorav de fleste datapunktene bør settes av til trening.

For å ta i bruk valideringsdataene, må treningsprosedyren vi implementerte tidligere (da vi så på klassifisering), modifiseres. Den bør regne ut relevante metrikker på valideringsdataene et gitt antall ganger i løpet av treningen, for eksempel etter hver tiende epoke (frekvensen er avhengig av kompleksiteten til oppgaven og hvor mye lagringsplass vi vil avse). Kode for en mulig treningsloop som gjør dette på lineær regresjon med stochastic gradient descent fra `sklearn` er vist under.

```
model = sklearn.linear_model.SGDRegressor(learning_rate='constant', eta0=0.01)
```

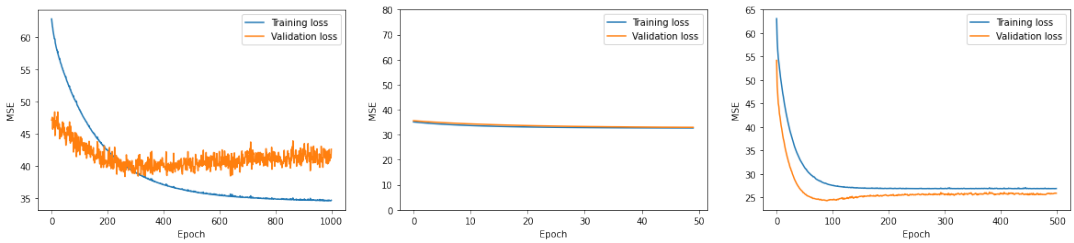
```
Train_losses, val_losses = [], []
n_epochs = 100
for epoch in range(n_epochs):
    model.partial_fit(X_train, y_train)

    # Monitor training loss
    y_train_pred = model.predict(X_train)
    train_loss = sklearn.metrics.mean_squared_error(y_train, y_train_pred)
    train_losses.append(train_loss)

    # Monitor validation loss
    y_val_pred = model.predict(X_val)
    val_loss = sklearn.metrics.mean_squared_error(y_val, y_val_pred)
    val_losses.append(val_loss)
```

Denne koden vil gi oss to arrays vi kan plotte enten underveis eller i etterkant av treningen, for å studere hvordan tapet utvikler seg på henholdsvis trenings- og valideringsdataene.

Studer figur 15a. Her tilpasses en modell av høy kompleksitet til et datasett, og siden tapet på valideringsdataene flater ut imens tapet på treningsdataene fortsetter å minke, kan vi konkludere med at modellen *overtilpasser*, sannsynligvis fordi den har for høy kompleksitet. Overtilpasningen gjør at modellen ikke klarer å generalisere på de usette datapunktene i valideringsdatasettet. Det motsatte ser vi i figur 15b. Her er tapet høyt på både trenings- og valideringsdataene, som tyder på at modellen har for lav kompleksitet for å tilpasse seg til variabiliteten i treningsdataene, men til gjengjeld generaliserer like godt til valideringsdataene gjennom hele treningsprosedyren. Se til slutt på figur 15c, hvor tapet minker jevnt på både trenings- og valideringsdataene, men flater ut likt for begge datasettene. Dette betyr at modellparametrene har oppnådd de mest optimale verdiene vi kan forvente i denne treningsprosedyren, men *uten* at disse er overtilpasset til treningsdataene.



(a) (b) (c)

Figure 15: Trenings- og valideringstap for tre ulike modeller, se beskrivelse i teksten.

For å oppsummere bør modellens ytelse monitoreres under treningen, gjerne i form av plott som viser ulike metrikker, inkludert tap, på trenings- og valideringsdataene.

- Hvis tap på treningsdataene synker jevnt, mens loss på valideringsdataene er høyere og/eller flater ut, overtilpasser modellen.
- Hvis begge kurvene flater ut med høy loss, undertilpasser modellen. Det betyr som oftest at den ikke har kapasitet eller riktig form til å tilpasse seg til dataene. Det mest åpenbare tegnet på undertilpasning er høy loss på treningsdataene.
- Hvis begge kurvene synker jevnt og flater ut, betyr det at modellen klarer å modellere treningsdataene, og samtidig klarer å generalisere til nye data.

Disse metrikkene kan også brukes til å justere hyperparametre underveis i treningen. For eksempel er det vanlig å minke læringsraten i løpet av treningen. De kan også brukes til å avgjøre når treningen bør stanse: det er ikke nødvendig å bestemme på forhånd hvor mange epoker modellen skal trene. I stedet kan man lage et *early stopping criterium* som stanser treningen for eksempel når tapet på valideringsdataene ikke har minket innenfor en toleranse i løpet av de siste n epokene.

1.4.5 Kryssvalidering

Oppgave: Hvordan velger vi hvilke datapunkter som havner i henholdsvis trenings-, test og valideringsdatasettet, og påvirker dette testresultatet?

Inndeling av datasettet i trenings-, test- og valideringsdata (skal) gjøres tilfeldig, men denne splitten kan påvirke testresultatet. Vi må altså forvente noe varians på testresultatet, avhengig av datasplitt. Dette er en utfordring vi må håndtere. I tillegg er det utfordrende at vi får færre datapunkter å trene på når vi setter av deler av dataene til validering og testing. Generelt gjelder at vi ønsker så mye treningsdata som mulig. En løsning på dette er *kryssvalidering*. Den enkleste former er såkalt *Leave-one-out cross validation* (LOOCV), hvor én observasjon fjernes fra det opprinnelige datasettet, og brukes til testing. Det opprinnelige datasettet består av n datapunkter, treningsdatasettet har størrelse $n-1$, og ett datapunkt brukes til testing. Valideringssettet lages ved å deretter hente ett eller flere datapunkter fra treningsdatasettet. Denne prosedyren gjentas $n$ganger: Trening på $n-1$ datapunkter og testing på ett datapunkt. Vi står da igjen med n testresultater, hvorav hvert er beregnet på ett datapunkt. Gjennomsnittet av disse n testresultatene er LOOCV-estimatet av metrikken vi ønsker å beregne. Siden estimatet har brukt samtlige tilgjengelige datapunkter til sammen, er det det beste estimatet vi kan lage.

LOOCV er en ressurskrevende prosedyre, da modellen må tilpsses like mange ganger som man har treningsdatapunkt. En mer utbredt metode er derfor *k-fold cross validation*, med k splitter i stedet for n. Man velger da en verdi for k, og splitter datasettet i k deler. Hvor hver iterasjon k spiller en annen del av datasettet rollen som testdata, mens resten brukes til trening, modellen tilpasses k ganger, og man oppnår k testresulater som brukes til å beregne det endelige estimatet av testresultatet.

Oppgave: Hva er fordeler og ulemper ved k-fold cross validation sammenliknet med LOOCV?

Ved bruk av kryssvalidering kan dere sjekke at likning 70 holder, ved å splitte dataene k (eller n) ganger, trene k modeller, beregne MSE, bias og varians for alle k splitter og modeller, og beregne gjennomsnittet av de k resulterende verdiene av MSE, bias og varians – for eksempel ved bruk av koden under.

```
avg_expected_loss = np.apply_along_axis(lambda x: ((x - y_test) ** 2).mean(),
                        axis=1, arr=all_pred).mean()
mean_predictions = np.mean(all_pred, axis=0)
avg_bias = np.sum((mean_predictions - y_test) ** 2) / y_test.size
avg_var = np.sum((mean_predictions - all_pred) ** 2) / all_pred.size
```

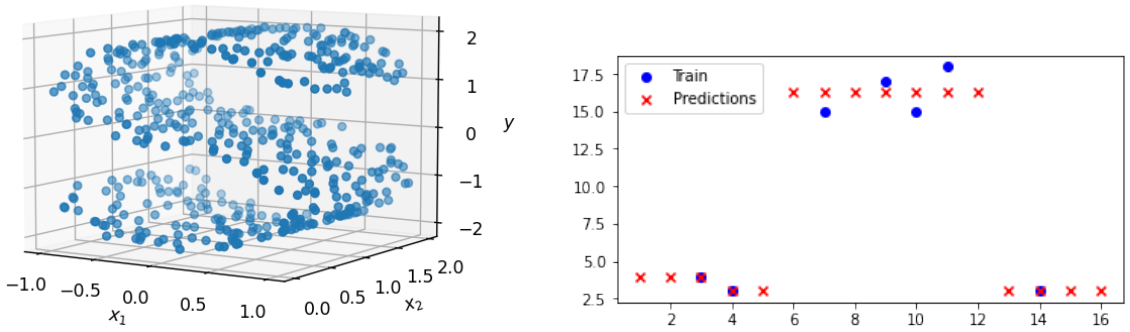
1.4.6 Regresjonstrær

Se på dataene i figur 16a. Her har vi to features x_1 og x_2, og et target y med en tydelig fordeling, men som vi likevel ikke kjenner funksjonsformen til.

Oppgave: Prøv gjerne å generere disse dataene selv ved bruk av `sklearn.datasets.make_s_curve()`, og se om dere klarer å tilpasse et polynom av valgfri grad til dataene.

Vi innser raskt at y ikke følger et polynom, og at regresjon ikke er er god vei til mål. I stedet kan vi innse ved å studere dataene at de befinner seg i flere regioner. Vi husker kanskje også at beslutningstrær er godt egnet til å splitte data til ulike regioner, gjennom ulike splittkriterier. Tidligere har vi brukt beslutningstrær til klassifisering, hvor data splittes gjennom treets noder til løvnoder som predikerer klassen til hver datainstans. Den samme modellstrukturen kan brukes i regresjon, siden *hver splitt i treet deler opp datarommet*. Beslutningstrær kan brukes til regresjon ved at de splitter datarommet til flere områder, og predikerer én verdi per område.

Et enkelt eksempel for intuisjon er vist i figur 16b. Her gjøres prediksjonene (de røde kryssene) av et lite beslutningstre med dybde 2, altså treet predikerer y-verdi basert på x-verdiene. Selv om dette problemet er enkelt, hadde det vært vanskelig å få til med en regresjonsmodell. Vi ser at treet har identifisert tre regioner med konstante prediksjoner. For de S-formede dataene i figur 16a forventer vi å trenge et dypere tre. Koden under kan brukes til å lage et vilkårlig dypt tre, tilpasse det på en del av dataene og teste det på de resterende dataene, og plotte resultatet. Da jeg kjørte koden fikk jeg et tre med dybde 34 og 375 noder.



(a) (b)

Figure 16: (a) Et S-formet datasett generert ved hjelp av `sklearn`, se beskrivelse i teksten, og (b) et datasett med tre regioner (blå punkter) og prediksjoner fra et beslutningstre (røde kryss).

```
X, _ = sklearn.datasets.make_s_curve(n_samples=500)
X_data, y_data = X[:,0:2], X[:,2]
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data)
regressor = tree.DecisionTreeRegressor().fit(X_train, y_train)
y_pred = regressor.predict(X_test)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(X_test[:,0], X_test[:,1], y_pred, label="Predictions")
ax.scatter(X_test[:,0], X_test[:,1], y_test, label="Labels")
```

Oppgave: Lag en liste over ulikheter mellom beslutningstrær (både for klassifisering og regresjon) og lineær/logistisk regresjon (med polynomisk feature extraction). Se tabell 7 når du gir opp.

Table 7: Forskjeller mellom beslutningstrær og regresjonsmodeller.

Trær	**Regresjon**
Feature scaling er ikke nødvendig. Beslutningstrær deler inn dataene i regioner, så det er ikke nødvendig å skalere dem.	Features bør ha samme størrelsesorden, helst $\mathcal{O}(1)$, så ikke tilpasningen av modellparametrene fører til store gradienter.
Hvis læringsalgoritmen ikke har begrensninger i tredybde, kan treet bli for dypt og overtilpasse.	Funksjonsformen er bestemt. Læringsalgoritmen kan ikke legge til ledd og lage en mer kompleks modell, eller fjerne ledd om mindre kompleksitet er tilstrekkelig.
Har en overordnet struktur, men den spesifikke oppbygningen – *arkitekturen* – bestemmes under trening.	Har en gitt likningsform med antall ledd, og parametrene til hver variabel og variabelkombinasjon tilpasses.

En viktig egenskap som både regresjonsmodeller og beslutningstrær deler er at de er *tolkbare*: For regresjonsmodeller vet vi at størrelsen til modellparametrene angir viktigheten til hver variabel (hvis β_4, hhv w_4 er stor, har x_4 stor innflytelse på prediksjonen. For beslutningstrær er splittkriteriene forståelige for mennesker, og vi skjønner at features som splittes tidlig er viktigere enn features som splittes senere. Det samme gjelder ikke for modelltypen vi skal se på i neste omgang.

1.5 Ensemble-modeller

Dette avsnittet er basert på Ruslan Khalitovs forelesning 16.09.2024.

Det kan være vanskelig å velge riktig modell for et gitt datasettet og problem. Generelt har alle modeller sine antakelser og svakheter, for eksempel antar lineær regresjon at forholdet mellom features

og target(s) er lineært, og beslutningstrær antar at datarommet kan separeres ved bruk av vertikale beslutningsflater (splitter). Tanken bak *ensemble learning* er at flere modeller kan kombineres, slik at de kompenserer for hverandres svakheter, og til sammen utgjør en samling, et *ensemble*, som benytter seg av hver enkelt modells styrke. Intuitivt kan vi se for oss en gruppe bestående av eksperter, der gruppens endelige beslutning tar alle ekspertenes vurdering med i beregning, men forkaster vurderingen som ikke deles av majoriteten.

For å lage et ensemble av modeller, kombineres flere modeller. Dette kan gjøres på ulike måter. Konseptuelt har vi tre ulike måter:

- **Parallellt**: Flere modeller trenes uavhengig av hverandre, og prediksjonene deres kombineres til en enkelt prediksjon
- **Sekvensielt**: Flere modeller kommer etter hverandre, og hver modell opphever feilen begått av foregående modell. Til sammen kommer rekken av modeller frem til én prediksjon, der hver modell har minimert feilen gjort av modellen før.
- **Hierarkisk**: Vi bruker en (eller flere) modell(er) til å kombinere prediksjonen fra en foregående parallellkombinasjon av flere modeller.

Når vi bruker en trent modell til å gjøre prediksjoner, sier vi at vi gjør *inferens*. Skillet mellom trening og inferens er viktig, og særlig synlig i ensemble learning.

Vi kan lage et parallellt ensemble på flere måter: Vi kan trene samme type modell (for eksempel et beslutningstre) med n ulike valg av hyperparametre. Da ender vi opp med n ulike modeller av samme type. På ett datapunkt gir disse modellene oss n ulike prediksjoner, som vi *aggregerer* til en endelig prediksjon. Alternativt kan vi trene n ulike modeller (for eksempel et beslutningstre, en lineær regresjonsmodell og et nevralt nettverk), som igjen gir oss n ulike prediksjoner vi aggregerer til ensemblets endelige prediksjon. Videre kan vi, i tillegg til å bruke ulike modeller og ulike hyerparametre, trene de ulike modellene i ensemblet på ulike deler av treningsdataene.

1.5.1 Bagging

Når vi lager et ensemble bestående av ulike modeller trent på ulike deler av de tilgjengelige treningsdataene og kombinerer prediksjonene fra disse til én prediksjon, må vi ta stilling til to spørsmål:

1. Hvordan bør vi aggregere de ulike modellenes prediksjoner?
2. Hvordan bør vi velge ut hvilke deler av treningsdataene vi trener hver enkelt modell på?

Aggregering av prediksjoner for regresjon gjøres oftest i form av et gjennomsnitt av alle modellenes prediksjoner. For klassifisering gjøres det oftest gjennom å predikere klassen som svarer til flertallet av enkeltmodellenes prediksjoner, altså en direkte avstemning, eller å gjøre en vektet avstemning mellom modellene.

Når det gjeler utvalg av treningsdata, er *bootstrapping* et sentralt konsept. Den underliggende tanken bak bootstrapping er at vi vet at vi ikke kan samle nok data til å representere den underliggende fordelingen bak et fenomen perfekt, men gitt et stort nok datasett kan vi få til et tilstrekkelig representativt utvalg. Likevel vil et representativt utvalg ikke uten videre fortelle oss om usikkerheten i estimatene vi gjør basert på disse dataene, altså hvor stor spredning det har. Gitt datasettet vi har samlet kan vi dog lage et estimat av spredning, eller usikkerhet, ved hjelp av teknikken *bootstrapping*. Dette går ut på å trekke flere datapunkter fra det samme datasettet *med tilbakelegging* (dette er viktig: det samme datapunktet kan finnes flere ganger i resulterende datasett), og slik ende opp med flere ulike datasett fra det éne datasettet vi startet med. Vi bruker disse ulike datasettene til å estimere den samme størrelsen flere ganger, og slik ende opp med en *fordeling* av estimatene. Denne fordelingen kan vi bruke til å beregne en spredning, eller usikkerhet, i estimatet vårt. Vi startet altså med ett datasett som vi kunne lage ett estimat fra, men har ved hjelp av bootstrapping skaffet oss en fordeling – uten å ha fått tilgang til flere datapunkter eller datasett. Vi har laget mer uten å måtte samle mer data, altså "pulled us up by our own bootstraps".

Begrepet *bagging* er satt sammen av **b** fra **b**ootstrap, og **agging** fra **agg**reger**ing**.

En enkel type ensemblemodell som bruker bagging er *random forest* (tilfeldig skog). Denne lages ved å sette sammen ulike beslutningstrær, eventuelt stumper. For at ensemblemodellen skal bli god, må de ulike trærne være *diverse* og *uavhengige*. Dette oppnår vi gjennom å trene trærne på ulike deler av dataene (bootstrapp-teknikken), og dessuten ulike utvalg av data-features, slik at trærne modellerer ulike sammenhenger. Til slutt aggreges prediksjonene fra alle trærne til én prediksjon – og *bagging* har skjedd.

Generelt har parallelle ensemblemodeller til felles at de består av modeller som trenes uavhengig av hverandre. Den endelige prediksjonen lages ved at de individuelle prediksjonene aggregeres, gjennom en type gjennosnitt eller avstemning. De individuelle modellene kan lages ved å bruke ulike modelltyper, ulike hyperparametre, ulike random seeds av samme læringsalgoritme, ulik feature engineering, ulike utvalg av treningsdataene, med mer.

1.5.2 Boosting

Innen maskinlæring refererer begrepet *boosting* til algoritmer som iterativt trener svake modeller (for eksempel trestumper) på en datafordeling, og kombinerer disse til en sterk modell (ensemblet). Konseptet stammer fra en samling publikasjoner av Kearns og Valiant (1988, 1989), og Schapire (1990), som undersøkte muligheten for at flere svake modeller, altså modeller hvis prediksjoner er kun svakt korellert med target i dataene, kan settes sammen til en sterk modell, altså en modell hvis prediksjoner er vilkårlig sterkt korellert med targets i dataene. Begrepet boosting handler om at feilene begått av én modell gjør den påfølgende modellen i iterasjonen bedre ("booster" den). I stedet for å kombinere flere modeller parallelt, organiseres de altså *sekvensielt*, og hver modell forholder seg til den forrige på en måte som gjør ensemblet sterkere enn hver enkelt modell er for seg selv. Vi skal se på to algoritmer som gjør dette.

AdaBoost-algoritmen bygger et ensemble av modeller som korrigerer hverandres feil, gjennom en iterativ treningsprosedyre. I starten av prosedyren har alle punktene i treningsdataene samme vekt, og vi trener én modell som predikerer på disse dataene. Basert på targets ser vi for hvilke datapunkter modellen har størst tap, og i neste iterasjon økes vektene for disse datapunktene. Deretter trenes en ny modell, som igjen predikerer på et datasett, før vektene igjen justeres.

Gradient boosting er, til forskjell fra AdaBoost, ikke basert på vekting av observasjoner. I stedet predikerer hver modell *forskjellen mellom targets og den forrige modellens prediksjon*, såkalte *pseudo-residuals*. Det nye ensemblet lages ved å følge læringsregelen

```
new_ensemble = previous_ensemble - learning_rate * new_tree
```

Dette uttrykket bør minne deg om gradient descent, som er opphavet til navnet *gradient* boosting. Vi gjør altså ikke gradient descent i rommet over alle mulige parameterverdier, men i rommet over alle mulige trær ensemblet vårt kan bestå av. Gradienten i dette tilfellet er altså `-(label-prediction)`.

Oppgave: Trebaserte ensemblemodeller du bør ha hørt om, og helst brukt på et datasett, er `CatBoost`, `LightGBM`, `AdaBoost`, og `XGBoost`.

Det er en god regel å alltid bruke en ensemble-modell som referanseverdi for hvor godt en modell kan gjøre det, når du jobber med et maskinlæringsproblem med tabulære data (altså den typen data vi bruker i dette kurset).

2 Nevrale nettverk

Når vi gjør maskinlæring ønsker vi å tilpasse en funksjon f som gir oss et estimat basert på x, i veiledet læring et estimat av target y, altså $\hat{y} = f(x)$. Vi har sett på flere måter å modellere f på:

- Med lineær/logistisk regresjon er funksjonsformen til f gitt, og treningen går ut på å tilpasse parametrene i funksjonen. Dette er eksempler på *parametrisk modellering*.

- Med beslutningstrær er ikke formen til f gitt, og treningen går ut på å bygge treet og velge splittkriterier, for å dele opp datarommet. Dette er et eksempel på *ikke-parametrisk modellering*.

I en regresjonsmodell multipliseres dataene med parametre og adderes deretter (linearitet). I beslutningstrær flyter dataene gjennom modellen uten å transformeres; hver node splitter dataene, men transformerer dem ikke. Vårt neste steg innebærer å lage ikke-lineære modeller som transformerer dataene.

2.1 Perceptron

Perseptronet er forgjengeren til moderne nevrale nettverk, og ble introdusert av McCulloch og Pitts (1943). Den første implementasjonen ble bygget, i hardware, av Rosenblatt (1957). Tanken var at perseptronet skulle være en maskin; ikke et program (denne setningen er verdt å dvele ved). Et perseptron består av én eller flere beregningsenheter, ofte kalt *noder*, og i den opprinnelige formuleringen var nodene *threshold logic units* (TLU). Disse mapper inputen x til en output $f(x)$ som tar en binær verdi,

$$f(\mathbf{x}) = \mathrm{H}(\mathbf{w}\mathbf{x} + b)\,, \tag{73}$$

hvor H er Heaviside stegfunksjonen, se figur 18a, og i vår kontekst kalles en *aktiveringsfunksjon*. Både $\mathbf{x}$ og $\mathbf{w}$ er vektorer i det generelle tilfellet. En TLU, beskrevet av likningen over, er enten *aktiv*, altså har output=1, eller *ikke aktiv*, altså har output=0. Hvilke data $\mathbf{x}$ som gir hvilken aktivering av noden er avhengig av vektene $(\mathbf{w}, b)$. Siden output er binær, gjør denne en klassifiseringsoppgave, og for at perseptronet skal ha høy treffsikkerhet i klassifiseringen må vektene justeres til riktig verdi. Dette gjøres gjennom veiledet læring.

Som før sendes instanser fra treningsdataene enkeltvis gjennom modellen, denne gjør en prediksjon $\hat{y}$ og tapet beregnes basert på target y. Parametrene i modellen oppdateres for å redusere tapet per treningsinstans. Dette gjentas for alle instansene i treningsdataene, hvilket utgjør én epoke (epoch), og hele prosessen gjentas flere ganger (epochs). Vi kan implementere et perseptron ved å gjenbruke koden fra logistisk regresjon, med likning 73 som modell, altså:

```
activation = sum(weight_i * x_i) + bias
prediction = 1.0 if activation >= 0.0 else 0.0
```

Igjen er treningsprosessen en loop over treningsdataene med egnet tapsfunksjon $\mathcal{L}$, og parametrene oppdateres etter regelen i likning 13, gjentatt under:

$$\theta^{t+1} = \theta^t - \eta \frac{\partial}{\partial \theta} \mathcal{L}(f(\mathbf{x}; \theta), y)\,. \tag{74}$$

Oppgave: Hva er problemet med denne prosedyren? Ikke les videre før du har prøvd å besvare dette spørsmålet. Hint: Hva skjer hvis du prøver å derivere tapsfunksjonen med $f(x)$ fra likning 73?

Når vi skal derivere tapsfunksjonen, som inneholder prediksjonen, er vi nødt til å derivere Heaviside-funksjonen. Denne er 0 overalt, og ∞ i ett punkt, og dermed ikke definert. Som dere garantert husker, må tapsfunksjonen være deriverbar for å finne gradienten til parameteroppdateringen. I den opprinnelige formuleringen av perseptronet (Rosenblatt) er leddet med den deriverte utelatt, så følgende uttrykk kan brukes

$$w_i \leftarrow w_i + \eta\,(y_i - \hat{y}_i)\ . \tag{75}$$

I kode er det vanlig å utvide $\mathbf{x}$ med et første element med verdi 1, og $\mathbf{w}$ har et tilsvarende ledd som representerer b, slik at vi får den mer kompakte operasjonen under.

```
w = w + learning_rate * (expected - predicted) * x
```

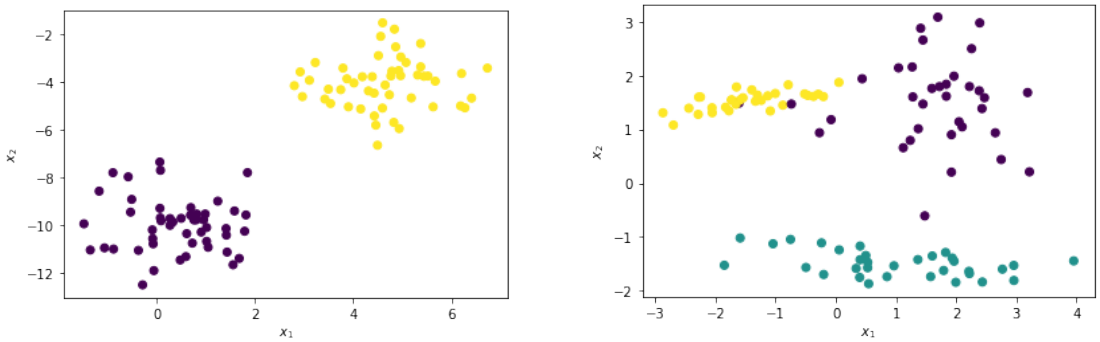
Se på dataene i figur 17a. Disse representerer to features x_1 og x_2, og fargen angir de to mulige klassene y. De er generert ved hjelp av `sklearn.datasets.make_blobs`, og for å tilpasse et perseptron som løser klassifiseringsoppgaven de representerer, kan vi bruke `sklearn.linear_model.Perceptron`, se under.

```
perceptron = Perceptron().fit(X_train, y_train)
y_pred = perceptron.predict(X_test)
```

Til info er dette en wrapper rundt `SGDClassifier` som vi har brukt tidligere, med `loss="perceptron"` og `learning_rate="constant"`. Vi finner de tilpassede parameterverdiene tilsvarende som for lineær regresjon:

```
ws = perceptron.coef_
bs = perceptron.intercept_
```

Se på dataene i figur 17b. **Oppgave:** Hvor mange features og klasser har dataene? Hvor mange vekter trenger et perseptron for å tilpasse dem? Det kan være nyttig å tegne en figur.



(a) (b)

Figure 17: Klassifiseringsdatasett med (a) to klasser, generert av `sklearn.datasets.make_blobs`, og (b) tre klasser, generert av `sklearn.datasets.make_classification`.

For å gjøre denne klassifiseringsoppgaven, kan vi gjenbruke koden fra tidligere, og treningsprosedyren forblir den samme. Hvis man har implementert `.fit()` og `.predict()` selv, må koden eventuelt tilpasses for å håndtere en vektmatrise (i stedet for en -vektor), da parameteroppdateringen nå følger:

$$w_{i,j} \leftarrow w_{i,j} + \eta\,(y_j - \hat{y}_j)\,x_i\,. \tag{76}$$

Her er

- $w_{i,j}$ vekten på forbindelsen mellom input i og node j,
- η læringsraten,
- $\hat{y}_j$ og y_j henholdsvis output og target for klasse j,
- x_i feature-verdi i for det aktuelle datapunktet.

Vi gjenbruker koden fra tidligere, og kan finne ut hvor mange klasser og features vi har (som vi strengt tatt bør ha kontroll på før vi tilpasser modellen), og hvor mange vekter modellen har, ved hjelp av følgende kode.

```
perceptron = Perceptron().fit(X_train, y_train)
ws = perceptron.coef_
bs = perceptron.intercept_
print("Features:", X.shape[1])
print("Classes:", len(list(set(y))))
print("Weights:", ws.shape)
print("Biases:", bs.shape)
```

Har vi to features og tre klasser, må modellen ha to input-noder med forbindelser til tre output-noder. Da får vi seks vekter i $\mathbf{w}$ og tre bias-verdier i $\mathbf{b}$.

Logiske operasjoner

Perseptroner kan brukes til å gjøre logiske operasjoner, med riktig valg av vekter. Hvis vi har én variabel x_1, kan et perseptron gjøre den logiske operasjonen `NOT` med $(w1, b) = (-1, 0.5)$. For to variabler x_1, x_2 får vi den logiske operasjonen `AND` med $(w_1, w_2, b) = (1, 1, -1.5)$, og `OR` med $(w_1, w_2, b) = (1, 1, -0.5)$.

Oppgave: Kan kan et perseptron gjøre `XOR`? Hvordan? Hint: Det holder ikke med én TLU; sett sammen en kombinasjon av `AND`, `NOT` og `OR`.

2.2 Aktiveringsfunksjoner

En ulempe ved perseptronet er at en liten endring i input kan føre til en stor endring i output. Dette skyldes Heaviside-funksjonen, som sender funksjonsverdien til enten 0 eller 1, se figur 18a. Et bedre alternativ hadde vært å gi nodene muligheten til å returnere kontinuerlige verdier, eventuelt i intervallet $[0, 1]$. Om vi erstatter Heaviside-funksjonen med en funksjon som returnerer kontinuerlige verdier, har vi laget den typen node vi finner i moderne nevrale nettverk. Vanlige aktiveringsfunksjoner er sigmoid-funksjonen (som vi brukte for logistisk regresjon), se figur 18b, softmax, og ReLU, se figur 18c. Disse aktiveringsfunksjonene brukes til ulike formål: Hvis modellen skal gjøre binær klassifisering, er det vanlig å ha en sigmoid-aktiveringsfunksjon i noden i nettverkets siste lag, for å sikre at modellens prediksjon havner i intervallet $[0, 1]$. Sigmoid-funksjonen er den samme som tidligere, men gjentatt her for enklere sammenlikning med de andre aktiveringsfunksjonene:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \,. \tag{77}$$

I en modell som gjør ikke-binær klassifisering er det vanlig å bruke en softmax-aktiveringsfunksjon i output-nodene:

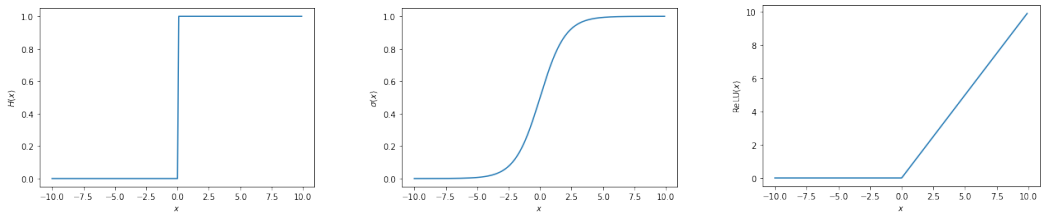
$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}} \,, \tag{78}$$

som sikrer at aktiveringene summerer til 1. Da representerer hver av output-nodene datapunktets predikerte tilhørighet til de respektive klassene, og vi må ha én output-node per klasse.

Moderne nevrale nettverk har som regel flere lag, mellom input- og output-lagene, se avsnitt 2.3. Slike modeller kalles multi layer perceptrons (MLP), da de har *multiple* lag. Vi står relativt fritt til å velge aktiveringsfunksjoner til disse lagenes noder, og i dette kurset vil vi holde oss til den såkalte rectified linear unit (ReLU) aktiveringsfunksjonen:

$$\text{ReLU}(x) = \max(0, x) \,. \tag{79}$$

MLP-modeller som ikke gjør klassifisering men regresjon, må ha output-noder som kan returnere kontinuerlige tallverdier som ikke er begrenset til et intervall (som $[0, 1]$ i klassifisering). Som regel brukes da en lineær aktiveringsfunksjon i output-laget.



(a) (b) (c)

Figure 18: De tre aktiveringsfunksjonene (a) Heaviside, (b) sigmoid, og (c) ReLU.

2.3 Arkitektur

Som nevnt over kan vi sette sammen noder på ulike måter for å lage nevrale nettverk. Hvordan nodene er satt sammen kalles nettverkets *arkitektur*. Nevrale nettverk består av *lag*, som igjen består av noder stablet i høyden, altså noder som ikke mottar input fra hverandre. Informasjon går kun mellom noder i ulike lag, altså ikke mellom noder i samme lag. Alle nevrale nettverk består av input-lag, output-lag, og indre/skjulte lag:

- **Input**-laget er det første laget i det nevrale nettverket. Dette mottar og sender data inn i nettverket. Det må derfor ha samme dimensjonalitet, dvs samme antall noder, som dataene har features.

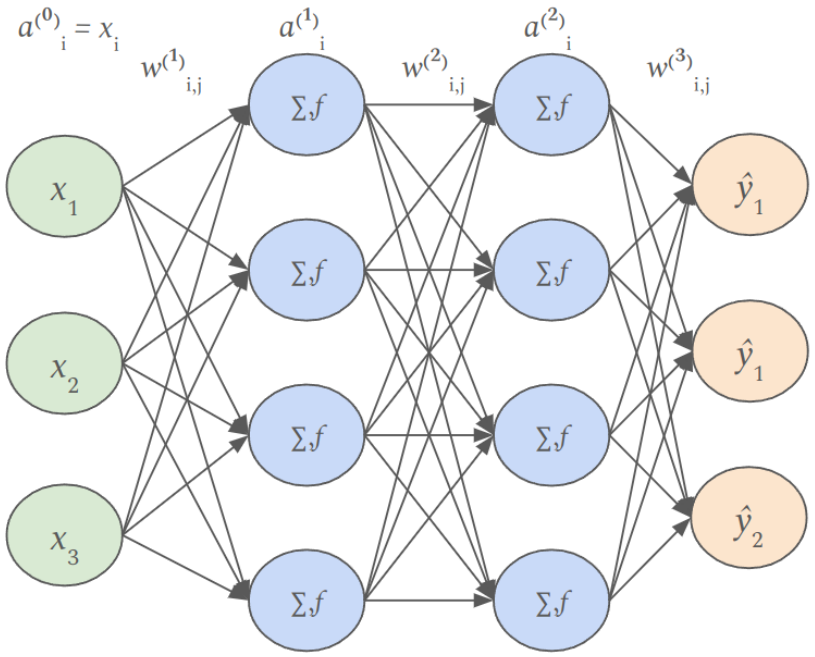


Figure 19: Skisse av et nevralt nettverk, som vist i forelesning.

- **Output**-laget er det siste laget i det nevrale nettverket. Dette representerer det nevrale nettverkets prediksjon. I tilfellet veiledet læring må dette laget ha samme dimensjonalitet, dvs samme antall noder, som targtets i dataene.
- **Indre** lag er alle lagene mellom input- og output-lagene. Disse omtales også som *skjulte* lag.

Akkurat som med valg av aktiveringsfunksjon, står vi fritt til å sette sammen nevrale nettverk med arkitekturen vi ønsker. Dette betyr ikke at hvilken som helst arkitektur er egnet for å løse problemet dataene våre beskriver. Ulike arkitekturer er bedre og dårligere egnet til ulike oppgaver, for eksempel er det vanlig å bruke konvolusjonslag til bildegjenkjenning, og transformer-blokket til sekvensielle data (som språk). Vi skal ikke se på slike arkitekturer i dette kurset, men holder oss til arkitekturer der alle nodene i hvert lag er koblet til alle nodene i det foregående og neste laget. Slike nevrale nettverk kalles som nevnt MLP'er, eller *fully connected feed-forward* nettverk. Her kommer *fully connected* nettop av at alle nodene i nabolag har forbindelser vil hverandre, og *feed forward* av at informasjon sendes kun fremover i nettverket, hvor *fremover* er definert som retningen fra input til output. La oss se nærmere på hva som skjer med dataene på veien fra input- til output-laget.

Bruk gjerne skissen i figur 19 til hjelp, eller lag din egen. Generelt har vi følgende uttrykk for aktiveringen a til en gitt node med indeks j i lag l av det nevrale nettverket

$$a^l_j = g\left(\sum_{i=0}^{n} w^l_{ij} a^{l-1}_i\right) . \tag{80}$$

Her er

- l indeks for lag, hvor indeks 0 angir input-laget
- i indeks for node i forrige lag
- j indeks for node i det aktuelle laget
- a^{l-1}_i aktivering av node i i forrige lag $l-1$
- g aktiveringsfunksjonen i det aktuelle laget.

For bedre intuisjon kan det være lurt å ta for seg én node og gå stegvis gjennom indeksene i likning 80.
Oppgave: Se på første (øverste) node i det første laget etter input-laget og skriv ned aktiveringen til

denne noden. Du bør komme frem til følgende uttrykk:

$$a_1^{(1)} = g\left(\sum_{i=0}^{n} w_{i1}^{(1)} x_i\right). \tag{81}$$

Vi tar for oss tilfellet der vi to input-features, og har bygget et skjult lag bestående av tre noder. Nettverkets videre arkitektur er uten betydning for den aktuelle diskusjonen. Vi har altså $(i = 1, 2)$, og $(j = 1, 2, 3)$. Vi kan skrive aktiveringene i nettverkets første indre lag som

$$a_1 = g\left(w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2\right) \tag{82}$$

$$a_2 = g\left(w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2\right) \tag{83}$$

$$a_3 = g\left(w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2\right) \tag{84}$$

etter å ha skrevet ut summen over input-laget. De tre likningene over kan skrives på matriseform som følger

$$\begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \end{bmatrix} = g\left(\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right). \tag{85}$$

Kontroller at matrisemultiplikasjonen på høyre side skjer mellom to matriser med dimensjoner henholdsvis 3×2 og 2×1, hvilket resulterer i en matrise av dimensjon 3×1, som er det vi har på venstre side av likhetstegnet.

Generelt består det første laget i nettverket av n noder, og det første indre laget av m noder. Overgangen fra input-laget til det første indre laget innebærer altså en transformasjon fra n input features til m nye verdier. Dette gjelder for alle lagene i et nevralt nettverk: De gjør en transformasjon av dataene de mottar, til et nytt datarom av samme dimensjon som antall noder i det aktuelle laget. På grunn av ikke-lineariteten til aktiveringsfunksjonen g, er transformasjonen ikke-lineær. *Oppsummert gjør hvert lag i det nevrale nettverket en egen ikke-lineær transformasjon av dataene.* Dette kan vi tolke som en automatisk feature-transformasjon som utvikles i takt med at det nevrale nettverket lærer fra data.

2.4 Backpropagation

Når nevrale nettverk gjør en prediksjon, altså produserer en output, sier vi at de gjør en *forward pass*: de går fra data (input) til prediksjon (output) ved å transformere dataene gjennom de ulike lagene. Hvis vektene i nodene, som svarer til det nevrale nettverkets modellparametre, har riktige verdier, er det små avvik mellom output og targets i dataene. Tenk over hvordan disse parametrene kan tilpasses, spesifikt: tenk over hvordan vi kan gjøre gradient descent for parametrene i et nevralt nettverk.

Som tidligere bør vi bevege oss i parameterrommet, i den retningen der tapet mellom prediksjon og targets minker, altså langs gradienten til tapsfunksjonen. Denne gradienten er en vektor med like mange elementer som vi har parametre i nettverket. Formelt er svarer dette til gradient descent for lineær regresjon, og igjen forteller størrelsen til hvert element i gradientvektoren oss hvor følsom tapsfunksjonen er for den tilsvarende parameteren. Forskjellen fra lineær regresjon ligger i at et nevralt nettverk bestående av mange lag, har mange noder der vi ikke kjenner den "riktige verdien". Dette gjelder alle nodene i indre/skjulte lag, da disse ikke er direkte forbundet med både dataene og targets. Likevel finnes det en metode for å estimere gradienten til tapsfunksjonen som funksjon av parametrene. Dette kalles *backpropagation*, eller tilbakepropagering, og i dette kurset forventes det at dere utvikler en intuisjon for hvordan backpropagation fungerer.

Vi bruker et eksempel for å få en intuitiv forståelse, og igjen anbefales det at du tegner en figur, gjerne inspirert av slides brukt i forelesningen. Anta at vi har et nevralt nettverk med to input-noder, to indre lag med henholdsvis fire noder, og et output-lag med tre noder. Anta videre at det nevrale nettverket har gjort en prediksjon $\hat{y} = (0.1, 0.1, 0.8)$ på et datapunkt med target $y = (1, 0, 0)$. Vi ser at både $\hat{y}_1$ og $\hat{y}_3$ har store avvik fra target-verdiene sine, mens $\hat{y}_2$ har et lite avvik fra sin tilsvarende target-verdi. Vi bruker indeks $l = 0$ for input-laget, $l = 1, 2$ for de to indre lagene, og $l = 3$ for output-laget. Vi ser

på aktiveringen til det første (øverste) nevronet i output-laget

$$\hat{y}_1 = a_1^{(3)} = g\left(w_{11}^{(3)} a_1^{(2)} + w_{21}^{(3)} a_2^{(2)} + w_{31}^{(3)} a_3^{(2)} + w_{41}^{(3)} a_4^{(2)}\right). \tag{86}$$

Her går altså indeks i i likning 80 over $i = (1, 2, 3, 4)$, mens $j = 1$, siden vi ser på den første noden, og $l = 3$, siden vi ser på det siste laget. I likningen over er det flere ting vi kan endre for å minke tapet: vi kan endre på parametrene b og $w_{i1}^{(3)}$ direkte, og aktiveringene av nodene i det tidligere laget, $a_i^{(2)}$, indirekte. Legg merke til at om vi endrer parametrene $w_{i1}^{(3)}$, må disse endres proporsjonalt med aktiveringene $a_i^{(2)}$. Vi ønsker å øke output $a_1^{(3)}$, som betyr at vi må styrke koblingen til aktive nevroner i lag 2, dvs nevroner med stor $a_i^{(2)}$. Enkelt sagt: Jo større a-ene er, jo større effekt har w-ene. Vi kan gjøre en tilsvarende analyse for $a_i^{(2)}$, for om alle $a_i^{(2)}$ som multipliseres med en positiv $w_{i1}^{(3)}$ ble sterkere, og alle $a_i^{(2)}$ som multipliseres med en negativ $w_{i1}^{(3)}$ ble svakere, hadde $a_1^{(3)}$ blitt større, som er det vi ønsker å oppnå. Enkelt sagt: Jo større w-ene er, jo større effekt har a-ene. Utfordringen er at vi ikke kan endre aktiveringene til indre lag direkte; vi kan kun endre dem indirekte gjennom parametrene deres. Vi må derfor endre på parametrene $w_{ij}^{(2)}$ tilhørende alle nodene i lag $l = 2$.

Før vi tar dette steget bakover i nettverket er det lurt å zoome ut, og huske på at vi kun har sett på én av verdiene i $\hat{y}$-vektoren: både $\hat{y}_2$ og $\hat{y}_3$ har sine egne preferanser for hvordan vektene og aktiveringene i de foregående lagene bør endres for at disse to delene av prediksjonen skal komme nærmere targets. Det er ikke gitt at $\hat{y}_1, \hat{y}_2$ og $\hat{y}_3$ er enige om hvilke endringer som bør gjøres, men vi samler uansett de ønskede endringene i lag $l = 2$ i en liste, før vi flytter oss et steg bakover i nettverket. Listen inneholder blant annet ønskede endringer for aktiveringene $a^{(2)}$, på samme måte som vi tidligere visste i hvilken retning vi ønsket å endre de ulike verdiene i $\hat{y}$, og basert på disse kan vi utføre samme analyse som vi gjorde under likning 86. Når vi har gjort dette for alle nodene i lag $l = 2$, lager vi en liste over ønskede endringer i lag $l = 1$, før vi tar et ytterligere steg tilbake i nettverket.

På dette tidspunktet har vi kommet til input-laget; det gir ikke mening å lage en liste over ønskede endringer i lag $l = 0$, siden dette er input-laget og aktiveringene svarer til verdiene i datasettet. Vi har derfor kommet til veis ende, ved å jobbe oss stegvis *bakover* i nettverket fra output, gjennom alle lagene, til input. Mer formelt har vi *propagert* feilen i prediksjonen *tilbake* til input, og gjennom denne prosessen funnet en liste over hvordan nettverkets parametre bør endre seg, altså *gradienten* vi trenger til parameteroppdateringen. Dette er essensen i backpropagation. Til slutt er det verdt å merke seg at dette må gjøres for alle datapunktene i treningsdatasettet; vi er ikke ute etter at modellen skal lære seg å predikere at $y = (1, 0, 0)$, men tilpasse prediksjonen til input-dataene. Akkurat som for lineær regresjon må vi altså sende alle treningsdataene (én epoch) gjennom nettverket, gjerne flere ganger (flere epochs).

2.5 Bygge, trene og predikere

Det finnes flere Python-biblioteker vi kan bruke for å lage nevrale nettverk, og i dette kurset anbefales `keras` og `tensorflow`. De grunnleggende byggesteinene i `tensorflow` er `keras`-operasjoner. For eksperimentering kan vi for eksempel lage en $(10, 3)$-dimensjonal tensor `x` fylt med ettall, med følgende kode.

```
from keras import ops
x = ops.ones((10,3))
```

Videre kan vi lage et enkelt nevralt nettverk, se figur i slides fra forelesningene, med to indre lag bestående av henholdsvis 6 og 7 noder, samt et output-lag bestående av 2 noder, som følger

```
from tensorflow.keras.models import Sequential()
model = Sequential()
model.add(Dense(6, activation="relu", name="layer1"))
model.add(Dense(7, activation="relu", name="layer2"))
model.add(Dense(2, activation="softmax", name="output"))
model(x)
```

Denne modellen heter `Sequential` fordi lagene kommer i en lineær sekvens, slik at dataene flyter sekvensielt fra input- til output-laget. Lagene heter `Dense` fordi alle nodene har koplinger til alle

nodene i forrige og neste lag. Koplingen er så tett (dense) som den kan være. Alle disse lagene initialiseres med tilfeldige verdier. Hvis vi derfor kaller denne modellen på tensoren `x` som vi laget tidligere, får vi derfor en $(10, 2)$-dimensjonal tensor med to ulike, men i utgangspunktet meningsløse, verdier gjentatt 10 ganger.

Ved bruk av koden over lager vi en `Sequential`-modell, og bruker operasjonen `add` for å legge til lag til modellen. Vi kunne tilsvarende gjort

```
model = Sequential(
    Dense(6, activation="relu", name="layer1"),
    Dense(7, activation="relu", name="layer2"),
    Dense(2, activation="softmax", name="output"),
)
model(x)
```

Her lager vi hele modellen direkte, ved å gi den en liste bestående av lag som argument. Vi kunne også gjort følgende

```
layer1 = Dense(6, activation="relu", name="layer1")
layer2 = Dense(7, activation="relu", name="layer2")
layer3 = Dense(2, activation="softmax", name="output")

layer3(layer2(layer1(x)))
```

Her lager vi sekvensen eksplisitt, uten å bruke `Sequential`. Siste lag evalueres rekursivt med input fra andre lag som argument, der input fra andre lag har input fra første lag som argument, og første lag har input `x` som argument. Merk at vi her ikke får et modell-objekt. Det viktige å huske her er at lag oppfører seg som elementer i en liste. Kommandoen `model.layers` kan brukes for å få tilgang til listen over lag den aktuelle modellen `model` består av. Et enkelt mentalt bilde av et nevralt nettverk i python er lag stablet oppå hverandre, eller *stacked layers*.

Merk at om vi kun lager modellen ved å stable lagene, men ikke evaluerer den (for eksempel på tensoren vår `x`), har den ingen vekter. Om vi kaller

```
model.weights
model.summary()
```

på en modell vi har bygget, men aldri evaluert, får vi ut en tom liste `[]` for vektene, og en tabell som viser `Output shape=?` for oppsummering av modellen. Modellparametrene finnes altså ikke. **Oppgave:** Hva er grunnen til dette?

Legg merke til at vi har kjørt modellen(e) vi laget over, på en tensor av shape $(10, 3)$, selv om ingen av modellens lag består av 3 noder. Grunnen er at modellens input er et objekt, ikke av typen `keras.layers`, så *input vises ikke som et lag*. For at modellen skal kjenne input-dimensjonen, og følgelig hvor mange parametre den består av, må vi enten evaluere den på en input-tensor, eller vi må spesifisere modellens input-shape. Dette kan gjøres som følger

```
model = keras.Sequential()
model.add(keras.Input(shape=(3,)))
model.add(Dense(6, activation="relu", name="layer1"))
```

Om vi kaller `model.summary()` p ådenne, vil vi se at den har output-shape 6, og 24 parametre. Vi kan også evaluere denne modellen på vår tensor `x` og sjekke shape ved hjelp av følgende kode

```
model(x).shape
```

Denne koden returnerer `TensorShape([10,6)]`, da input-tensoren vår består av 10 instanser, og modellens nåværende output-lag består av 6 noder. Merk at dette følger vanlig matrisemultiplikasjon, og tegn gjerne opp tensordimensjonene om du er usikker. Vi kan fortsette å legge til lag helt til vi er fornøyde med modellen – eller bare bygge en modell med antallet lag og noder vi ønsker, fra starten. Merk at shape på modellens output alltid svarer til antall noder i modellens siste lag.

Når modellen er bygget, kan vi kompilere og trene den. Når modellen kompileres velger vi hvilken metode som skal brukes for å optimalisere parametrene under trening, hvilken tapsfunksjon som skal brukes, eventuelle metrikker som skal beregnes underveis, med mer. I dette kurset bruker vi `operimizer=keras.optimizers.SGD()`, og tapsfunksjonen må passe til oppgaven og dataene. Modellen kan tilpasses som følger

```
history = model.fit(x_train, y_train, batch_size=64,
           epochs=10, validation_data=(x_val, y_val))
```

Her lagres beregnede metrikker i en `history`-variabel, hvert steg i SGD bruker 64 datapunkter, modellen trenes i 10 epoker, og den evalueres på valideringsdata underveis. Vi kan for eksempel studere hvordan tapet utvikler seg på trenings- og valideringsdataene ved bruk av følgende kode

```
train_losses = history.history["loss"]
val_losses = history.history["val_loss"]
epohcs = np.arange(1, len(losses)+1)
plt.plot(epochs, train_losses, label="Train loss")
plt.plot(epochs, val_losses, label="Validation loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

Som før er det lurt å stanse treningen når både trenings- og valideringstapet flater ut, men før tap på treningsdataene blir mye mindre enn tapet på valideringsdataene, siden dette er et typisk tegn på overtilpasning. `keras` gir muligheten for et `early stopping`-kriterium, hvor vi må angi hvilket kriterium som skal monitoreres, vanligst `monitor="val_loss"`, og hvor tålmodig læringsalgoritmen skal være før treningsprosessen stoppes.

Prinsipielt jobber vi med nevrale nettverk som med andre maskinlæringsmodeller, men vi har en større frihet i valg av arkitektur og antall parametre. Det krever trening å lære seg hvilke arkitekturer som er egnet for hvilke læringsoppgaver og datastrukturer. Oppsummert koker treningsprosessen av nevrale nettverk ned til følgende steg:

- Modellen bygges, og kompileres med valg av optimizer, tapsfunksjon, early stopping, med mer.
  `model.compile()`
- Når modellen er bygget kan vi trene den.
  `model.fit()`
- Deretter kan vi bruke den til å predikere, altså gjøre inferens.
  `model.predict()`
- Når modellen er trent, evaluerer vi den på egnet metrikk.
  `model.evaluate()`
- Når vi går hjem for dagen kan vi lagre modellen med
  `model.save('sted/for/lagring/navn.keras')`
- ...og laste den opp igjen neste morgen.
  `model = keras.models.load_model(sted/for/lagring/navn.keras)`.

3 Uveiledet læring

Vi har fremdeles som mål å lage databaserte modeller ved hjelp av maskinlæring, men går nå over til tilfellet der vi ikke har data som forteller oss hva som er *riktig svar* i modelleringsoppgaven. Da snakker vi om *unlabeled data*, altså data uten targets. Uveiledet læring – unsupervised learning – handler om å gjøre maskinlæring med læringsalgoritmer som finner mønstre i data uten targets. Vi skal se på tre kategorier uveiledet læring:

- Clustering / klynging

- Dimensionality reduction / dimensjonsreduksjon
- Outlier detection / anomalideteksjon

3.1 Clustering

Clustering, altså å lage klynger, kan tolkes som en form for "klassifisering uten labels", og formålet er å finne egnede grupperinger av like datapunkter i et datasett. Vi har flere mulige valg av likhetsmetrikk, for eksempel avstand mellom datapunktene. For å kunne gjøre en optimaliseringsprosess, som er grunnleggende i maskinlæring, trenger vi et læringssignal, altså et tap å minimere. I forelesningene ser vi på to ulike algoritmer: *k-means*, hvor likhet defineres som avstand, og *DBSCAN*, hvor likhet defineres som tetthet.

3.1.1 k-means

Den uveiledede læringsalgoritmen k-means deler datasettet bestående av n datapunkter, inn i k klynger. Hvert datapunkt tilordnes klyngen hvis klyngesentrum er nærmest datapunktet. Stegvis gjør denne algoritmen følgende

1. Velg et tall k, som representerer antall klynger.
2. Velg k tilfeldige klyngesentroider. En sentroide er midten av en klyngen, altså ikke et datapunkt (selv om den kan sammenfalle med et faktisk datapunkt).
3. Tildel hvert datapunkt en tilfeldig klynge.
4. Beregn den Euklidske avstanden mellom hvert datapunkt og alle sentroidene.
   $d^2(p,q) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2$
5. Tildel hvert datapunkt den nærmeste sentroiden fra steg 4.
6. Velg nye sentroider ved å regne ut middelverdien (`np.mean`) for hver klynge.
7. Gjenta steg 4, 5 og 6 inntil klyngene ikke endrer seg.

Utfordringen med denne algoritmen er at sannsynligheten for å velge tilfeldige sentroider (i steg 2) på uegnede steder, altså på en slik måte at algoritmen ikke klarer å danne gode klynger, øker eksponensielt med k. Under de to antakelsene at vi 1) velger riktig verdi for k, og 2) at hver sanne klynge har et likt antall datapunkter, kan vi anslå sannsynligheten for at de k sentroidene initialiseres i unike klynger, altså slik at hver sanne klynge har bare én sentroide. Vi husker at antallet måter å sortere k ulike elementer er $k!$, mens antallet måter å trekke k elementer fra en mengde på k elementer er k^k. Da går sannsynligheten for at hver sentroide tilhører en unik klynge som

$$\frac{k!}{k^k} = \frac{1 \cdot 2 \cdot 3 \ldots k}{k \cdot k \cdot k \ldots k} = \frac{1}{k} \cdot \frac{2}{k} \cdot \frac{3}{k} \cdots \frac{k}{k} \,. \tag{87}$$

Stirling's approksimasjon lar oss skrive om fakultetet som

$$k! \approx k^k e^{-k} + \mathcal{O}(\ln k) \,. \tag{88}$$

Innsatt i likningen over får vi at sannsynligheten for å initialisere k sentroider i unike klynger går som e^{-k}, altså eksponensielt avtakende med k.

En mer moderne løsning på initialiseringen er derfor å plassere den første sentroiden tilfeldig, og plassere neste sentroide på datapunktet som er lengst borte fra den første, gjentatt til alle de k sentroidene er plassert. Generelt: sentroide j initialiseres på datapunktet der minste avstand fra den forrige sentroiden er størst. **Oppgave:** Begge måtene å plassere sentroider er visualisert for ulike datafordelinger på www.naftaliharris.com/blog/visualizing-k-means-clustering/. Det kan være lurt å leke litt med dette visualiseringsverktøyet for å få en følelse av hvordan k-means fungerer.

Til slutt skal vi se på `kmeans++`, som bruker en mer avansert måte å velge plasseringene til sentroidene. I stedet for å velge sentroide j på datapunktet lengst borte fra forrige sentroide, velges datapunktet med en viss sannsynlighet, der sannsynligheten er proporsjonal med kvadrert avstand fra forrige sentroide. Grunnen til at dette er lurt, er at strategien der datapunktet lengst borte fra forrige sentroide velges

som sentroide j, er at det som oftest vil plassere sentroiden i utkanten av en klynge. Med sannsynlighetstilnærmingen velges fremdeles et datapunkt som er langt borte fra forrige klynge – proporsjonal med kvadrert avstand –, men med en viss tilfeldighet. Denne tilfeldigheten øker sannsynligheten for å havne nær midten av den faktiske klyngen. Det finnes et bevis på at denne fremgangsmåten for plassering av sentroider er forventet å være suboptimal med maksimalt en faktor $\log(k)$.

Vi kan enkelt lage en k-means-modell ved hjelp av `keras.cluster.MiniBatchKmeans`. Vi kan angi k gjennom argumentet `n_clusters`, og velge initialiseringsmetode for klyngesentrene gjennom argumentet `init`. Hvis vi ikke oppgir en verdi for k, vil algoritmen selv prøve å velge en optimal verdi for k. Dette kan fungere godt, men vil vel så ofte fungere mindre godt. *Valg av k er blant de store utfordringene innen clustering*, og det finnes flere metrikker for å måle kvaliteten på clusteringen. Biblioteket `sklearn.metrics.cluster` har flere metrikker innebygget, og dere bør ha hørt om tre av dem:

- **Inertia**: Sum av kvadrert avstand mellom punkter og sentroide. Lav verdi betyr kompakte klynger, som er bra. Svakheten ved denne metoden er at den letteste måten å oppnå en lav verdi, er ved å plassere en sentroide på hvert datapunkt, altså ha størst mulig k.
- **Silhouette Coefficient**: $\frac{b-a}{max(a,b)}$, hvor a angir midlere avstand innad i klyngen, såkalt mean intra-cluster distance, og b angir midlere avstand til nærmeste klynge, såkalt mean nearest-cluster distance. Den måler altså hvor nært et datapunkt er til sin egen klynge, sammenliknet med andre klynger. Den beste verdien til denne metrikken er 1, som betyr at punktet er nærmere sin egen enn andre klynger, 0 betyr at klyngene overlapper eller at punktet er på randen mellom to klynger, og den dårligeste verdien er -1, som representerer at datapunktet er i feil klynge.
- **Calinski Harabasz score**: Ratio mellom sum av kvadrert avstand mellom klynger, hvor en høy verdi er bra, og sum av kvadrert avstand mellom punkter i samme klynge, hvor en lav verdi er bra. Den måler altså hvor langt det er mellom klynger normalisert til hvor store klyngene er. En høy verdi av denne metrikken representerer tette klynger med god separasjon.

Med utgangspunkt i et datasett uten labels og hvor vi ikke klarer å se hvilken verdi av k vi bør velge, kan det være nyttig å tilpasse flere k-means-modeller med ulike hyperparametre, og regne ut flere metrikker for å se om vi klarer å identifisere optimale verdier.

Som vi ser eksempler på i forelesningene, finnes det tilfeller der k-means ikke klarer å identifisere klynger selv om vi oppgir korrekt verdi av k. Dette skjer typisk når avstanden innad i klyngen er større enn mellom naboklynger. Dette er en konsekvens av at k-means bruker avstand som et mål på likhet mellom punkter. Dette peker på en viktig detalj, som gjelder i all maskinlæring: *Alle læringsalgoritmer har implisitte antakelser om problemet de skal løse.* **Oppgave:** Hvilken implisitt antakelse ligger i k-means?

Generelt har k-means følgende ulemper

- følsom for antall sentroider valgt, automatisk valg fungerer ofte dårlig, og manuelt valg er kun så godt som utvalgsmetoden,
- svak med outliers, siden disse kan trekke sentroidene vekk fra de faktiske klyngene,
- fungerer dårlig i høydimensjonale rom fordi den baserer seg på Euklidsk avstand, som konvergerer til en konstant for store d,

og følgende fordeler

- lett å forstå og implementere,
- fungerer godt på store datasett (store n).

3.1.2 DBSCAN

DBSCAN ble utviklet av Martin Ester, Hans-Peter Kriegel, Jörg Sander og Xiaowei Xu i 1996, og er en uveiledet maskinlæringsalgoritme. Den baserer seg på å identifisere tre typer punkter i dataene:

- Kjernepunkter: datapunkter som ligger nært midten av en klynge.
- Ikke-kjernepunkter: datapunkter som tilhører klyngen, men ligger i utkanten.

- Outliers: datapunkter som ikke tilhører noen klynger. *En en hovedstyrke ved metoden at den ikke prøver å putte disse punktene i klynger.*

Relevante hyperparametre for DBSCAN er ϵ, som angir maksimal avstand mellom to punkter som kan tilhøre samme klynge, og `min_samples`, som angir minimum antall naboer et punkt må ha for å telle som et kjernepunkt. DBSCAN-algoritmen bruker følgende prosedyre:

1. Identifiser kjernepunkter ved å telle antall naboer innenfor en avstand ϵ. Hvis dette antallet er større enn `min_samples`, er datapunktene kjernepunkter.
2. Gå gjennom kjernepunktene og spre klyngen til datapunkter i nærheten. Hvis punktet er et kjernepunkt, bruk det til å utvide klyngen. Hvis ikke er datapunktet et ikke-kjernepunkt.
3. Identifiser alle punkter som ikke tilordnes en klynge, som outliers.

Vi vet altså om et datapunkt er et kjernepunkt basert på hyperparametrene, allerede i steg 1. Etter denne prosedyren vet vi også forskjellen mellom outliers og ikke-kjernepunkt. Ikke-kjernepunkter har færre enn `min_samples` naboer innenfor en avstand ϵ, men de er innenfor en avstand ϵ av et kjernepunkt. Disse punktene er del av klyngen, men de bidrar ikke til å utvide klyngen.

Fordeler ved DBSCAN er at vi ikke trenger å spesifisere antall klynger, at algoritmen heller ikke gjør en antakelse om dette antallet, og at algoritmen har en metode for å identifisere outliers og dermed er robust for outliers. Ulempene er følsomhet for valg av ϵ og `min_samples`, og dessuten at disse er globale variabler: det kan hende at vi får datasett der klyngene har ulik tetthet for ulike datapunkter. Siden vi bare kan velge én verdi av henholdsvis ϵ og `min_samples`, er DBSCAN mindre egnet i tilfeller der klyngetettheten varierer. DBSCAN er også tregere for store datasett (stor n), fordi den må beregne avstander mellom samtlige punkter for å finne naboer og identifisere klyngepunkter, og er svakere i høydimensjonale data, fordi den baserer seg på et mål av avstand, som er utfordrende i høydimensjonale rom, på grunn av dimensjonsforbannelsen.

I forelesningen går vi stegvis igjennom to eksempler der DBSCAN identifiserer klynger. **Oppgave:** Prøv selv med verktøyet https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/.

3.1.3 Hovedkategorier

Vi kan dele clustering-algoritmer inn i følgende fire hovedkategorier:

- **Sentroidebasert**: Starter med sentroider, assosierer datapunkter med disse og tilpasser frem til konvergens.
- **Tetthetsbasert**: Klynger defineres basert på tettheten til datapunkter. Punkter som ligger nærme hverandre defineres som klynger, mens isolerte punkter defineres som outliers.
- **Fordelingsbasert**. Gjør antakelsen at hver klynge består av punkter fra en sannsynlighetsfordeling, vanligvis en Gauss. Målet er å finne parametrene til denne fordelingen.
- **Hierarkibasert**. Lager et hierarki av klynger. I starten er alle datapunktene en egen klynge, og på slutten tilhører alle datapunktene samme klynge. Klyngene som dannes underveis sorteres i et hierarki, og man velger ut de genererte klyngene fra et steg i hierarkiet.

Oppgave: Hvilke av disse kategoriene hører k-means og DBSCAN til? De øvrige to kategoriene er ikke en del av pensum, i den forstand at dere bør ha hørt om dem, men ikke trenger å kjenne til eller kunne beskrive læringsalgoritmer fra disse to kategoriene.

3.1.4 Dimensjonsforbannelsen igjen

For intuisjon ser vi for oss at vi sampler tilfeldige punkter fra enhentskuben, altså generelt fra $[0, 1]^d$ i d dimensjoner. Slik kan vi fylle opp d-dimensjonale arrays som angir koordinater i det d-dimensjonale rommet, hvor vi havner i intervallet $[0, 1]$ langs alle d akser. Vi gjentar dette to ganger, slik at vi får koordinater til to punkter i det d-dimensjonale rommet, og så beregner vi den Euklidske avstanden mellom punktene. Dette kan vi gjøre for flere ulike valg av d, og så ofte vi orker. Koden nedenfor gjentar prosedyren 1000 ganger for seks ulike verdier av d, og resultatet er vist i figur 20.

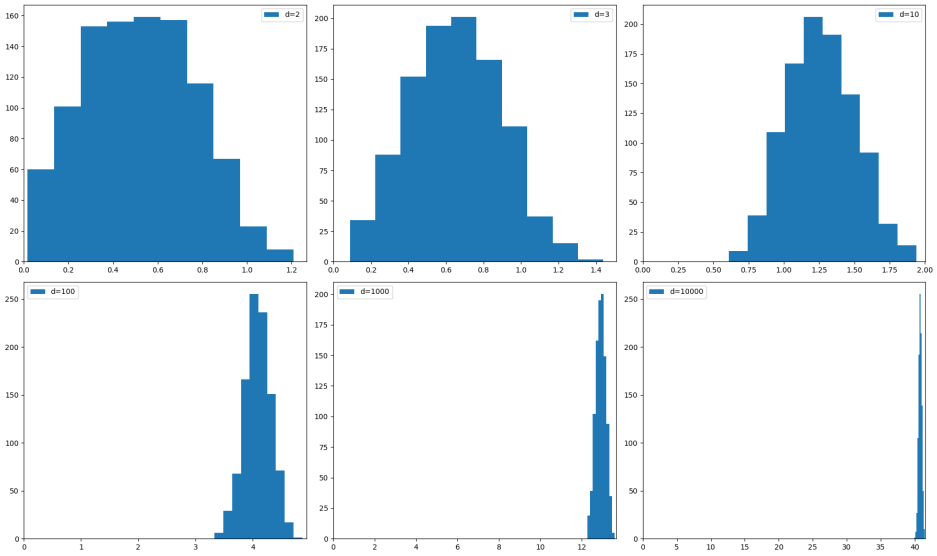


Figure 20: Fordelingen av Euklidske distanser mellom to punkter i en d-dimensjonal enhetskube.

```
def get_cube_dist(dim):
    dims = range(dim)
    P1 = np.array([np.random.uniform() for _ in dims])
    P2 = np.array([np.random.uniform() for _ in dims])
    dist = np.linalg.norm(P1-P2)
    return dist

n_iter = 1000
grid = [2, 3]
dims = [2, 3, 10, 100, 1000, 10000]
fig, axs = plt.subplots(grid[0],grid[1])

_i = 0
for _x in range(grid[0]):
    for _y in range(grid[1]):
        dim = dims[_i]
        _i += 1
        dists = [get_cube_dist(dim) for _ in range(n_iter)]
        axs[_x,_y].hist(dists, label=f"d={dim}")
        axs[_x,_y].legend()
        axs[_x,_y].set_xlim(left=0)
plt.show()
```

Oppgave: Hva forteller fordelingene i figur 20 oss om hvordan avstand oppfører seg i høydimensjonale rom?

Det finnes flere liknende øvelser vi kan gjøre for å få intuisjon. I slides fra forelesningen vises hvor stor andel av en d-dimensjonal enhetskube eller -kule som ligger i et tynt ytre skall av tykkelse ϵ. Basert på disse enkle øvelsene forstår vi at avstander skalerer eksponensielt med d. Dette gir opphav til det såkalte *Concentration of Distances*-fenomenet, hvis direkte konsekvens er at *relative* avstander blir mindre i høye dimensjoner. Avstandene i seg selv blir veldig store, mens *forskjellene* mellom avstander blir mindre. Dette vises tydelig i figur 20: fordelingenes middelverdier blir store med økende d, mens spredningene blir små. Tidligere, i diskusjonen om veiledet læring, forstod vi at dette betyr at mesteparten av datarommet er tomt når vi samler inn data med flere features enn $\mathcal{O}(10)$. I den nåværende diskusjonen forstår vi at konsekvensene også inkluderer at klynger i høydimensjonale rom ikke kan være kompakte. Clustering-algoritmer er avhengige av at datapunkter befinner seg i nærheten

av hverandre på en eller annen måte, hvilket generelt ikke vil være tilfelle i høydimensjonale rom. For å omgå dette problemet vil vi i neste omgang se på teknikker vi kan bruke for å redusere dimensjonaliteten i data. Dimensjonsreduksjon kan være både en preprosesseringsteknikk for clustering, og en dataanalysemetode i seg selv.

3.2 Dimensjonsreduksjon

Å redusere dimensjonaliteten i data kan være lurt av flere grunner:

- **Datavisualisering:** Vi kan ikke plotte høydimensjonale data, og mønstre, klynger, anomalier etc er ofte ikke synlige i høye dimensjoner, på grunn av dimensjonsforbannelsen.
- **Støyfjerning:** Å hente ut extracted features som representerer variansen eller strukturen i dataene best, kan fungere som støyfiltrering. Dette kan gi mer robuste (og tolkbare) modeller.
- **Feature discovery:** Metoder som oppdager strukturer i data som bevares når dimensjonaliteten reduseres, kan gi datasett med færre, sammensatte features som representerer den viktigste informasjonen fra de opprinnelige dataene. Dette hjelper ikke bare for å forstå dataene bedre, men gjør dem billigere å lagre og analysere.

Vi kan bruke uveiledet læring til å redusere dimensjonaliteten i data, altså å gå fra at dataene har et antall features f, til at de har et mindre antall features f'. Uveiledet læring kan brukes til dimensjonsreduksjon, og formålet er å bevare mest mulig av informasjonen fra den høydimensjonale i den laveredimensjonale versjonen av dataene. I forelesningene skal vi se på to etablerte metoder: PCA og (t-)SNE. Utover det finnes det, som ellers, mange flere metoder og biblioteker dere kan bruke, så lenge dere forstår hva de gjør.

3.2.1 Principal component analysis (PCA)

Metoden principal component analysis (PCA) ble utviklet av Karl Pearson i 1901, altså lenge før maskinlæring eller kunstig intelligens var etablert som fagfelt. Metoden går ut på å identifisere *aksene som maksimerer variansen i et datasett*. Disse aksene er hovedkomponentene, altså *the principal components* (PC), i dataene. De er lineære kombinasjoner av de opprinnelige featurene i dataene, og representerer uavhengige (ortogonale) akser.

Intuitivt kan vi se for oss at vi har et datasett med tre features: høyde, vekt og alder til en gruppe mennesker. Disse egenskapene vil ha noen korrelasjoner (som at høye mennesker veier mer, veldig unge mennesker veier mindre og er lavere, osv). PCA finner aksene, altså hovedkomponentende, tilsvarende de uavhengige retningene der dataene varierer mest. Den første hovedkomponenten, PC1, er retningen hvor dataene varierer mest, altså inneholder mesteparten av informasjonen. Den neste hovedkomponenten, PC2, inneholder nest mest informasjon, og den er ortogonal til PC1, for å sikre at den inneholder ny, uavhengig informasjon. At komponenter er ortogonale gir en garanti for at de inneholder ulik informasjon. Enkelt fortalt er PCA en metode som finner nye akser for dataene, hvor hver akse (PC) fanger ulike aspekter av variansen i dataene, uten å gjenta eller overlappe med de andre aksene (PC'ene).

Mer formelt er PC'ene egenvektorene til dataenes kovariansmatrise. Denne setningen er lettere å forstå om man gjør en beregning for hånd. Tabell 8 viser $N = 5$ datapunkter for $n_f = 2$ features. Gjennomsnittsverdiene er $\bar{x}_1 = 3$ og $\bar{x}_2 = 5.82$, og dette kan vi bruke til å beregne kovariansen mellom de fire parene (x_1, x_1), (x_1, x_2), (x_2, x_1) og (x_2, x_2) ved bruk av følgende formel,

$$cov(x_i, x_j) = \frac{1}{N-1} \sum_{k=1}^{N} (x_{ik} - \bar{x}_i)(x_{jk} - \bar{x}_j) \ , \tag{89}$$

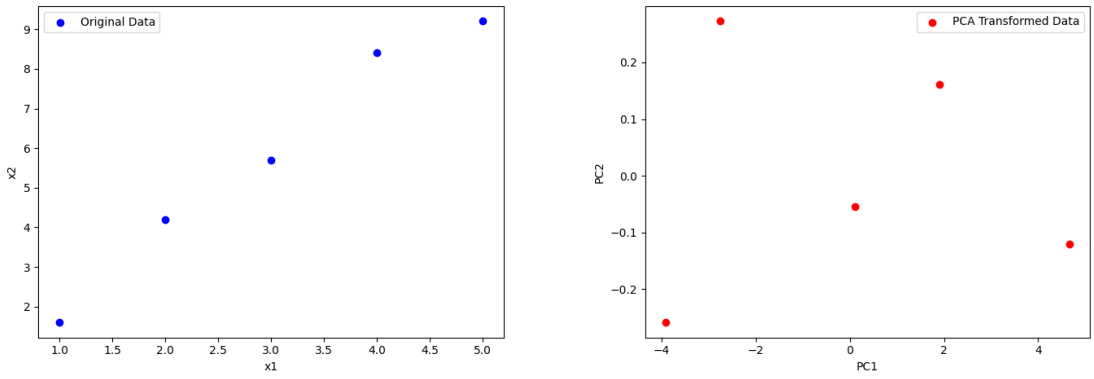
som er symmetrisk i i, j. Generelt har vi n_f^2 par for n_f features, og for våre to features blir kovariansmatrisen

$$S = \begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) \\ cov(x_2, x_1) & cov(x_2, x_2) \end{bmatrix} = \begin{bmatrix} 2.5 & 4.85 \\ 4.85 & 9.63 \end{bmatrix} . \tag{90}$$

Vi finner egenverdiene til denne matrisen ved å løse likningen $\det(S - \lambda I) = 0$, og de tilsvarende egenvektorene $e_{1,2}$ ved å løse $Se_i = \lambda e_i$, for $i = 1, 2$. **Oppgave:** Do it.

Table 8: Enkelt eksempeldatasett brukt i PCA-diskusjonen.

$\mathbf{x}_1$	$\mathbf{x}_2$
1	1.6
2	4.2
3	5.7
4	8.4
5	9.2



(a) (b)

Figure 21: Et datasett med (a) features x_1 og x_2, og (b) hovedkomponenter PC1 og PC2.

Dataene er plottet i figur 21a, og vi ser enkelt at mesteparten av variansen i dataene kan beskrives av en rett linje fra origo, mens det vil være litt varians i dataene rundt denne linjen (dette er vist tydeligere i figuren i slides fra forelesning). Vi kan altså lage to nye, ortogonale akser, dvs hovedkomponenter, som bevarer denne variansen, og transformere dataene til det nye koordinatsystemet beskrevet av disse, se figur 21b. Disse nye aksene er nettopp egenvektorene du akkurat beregnet.

I stedet for å gjøre PCA-analysen for hånd, kan vi bruke `sklearn`. Koden under bruker dataene fra tabell 8, finner to hovedkomponenter, og plotter de opprinnelige samt transformerte dataene.

```
from sklearn.decomposition import PCA
x1 = np.array([1,2,3,4,5])
x2 = np.array([1.6,4.2,5.7,8.4,9.2])

pca = PCA(n_components=2)
X = np.vstack((x1, x2)).T
X_pca = pca.fit_transform(X)
PC1 = [_x[0] for _x in X_pca]
PC2 = [_x[1] for _x in X_pca]

plt.scatter(x1, x2, color='blue', label='Original Data', marker='*')
plt.scatter(PC1, PC2, color='red', label='PCA Transformed Data')
plt.legend()
plt.show()
```

PCA er altså en transformasjon av de opprinnelige aksene til hovedkomponenter (egenvektorene), som beskriver variansen i dataene uavhengig av hverandre. Etter denne transformasjonen kan PC'ene sorteres i synkende rekkefølge, slik at de første bevarer mest av variansen. Ved dimensjonsreduksjon fra d til d' dimensjoner beholder vi kun de første d' aksene. Resultatet er at dataene projiseres til et laveredimensjonalt rom mens *mest mulig* av variansen til dataene bevares.

I koden over har vi transformert dataene til rommet spent ut av aksene PC1 og PC2, men ikke redusert dimensjonaliteten. For å gå fra to til én dimensjon, beholder vi kun den første hovedkomponenten,

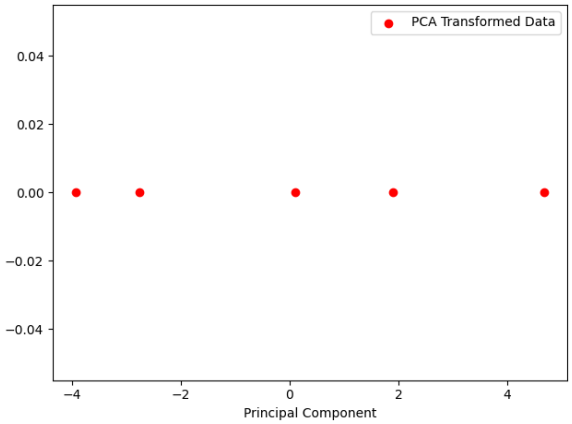


Figure 22: PCA av dataene i figur 21a til én hovedkomponent.

PC1. Alternativt kan vi direkte gjøre PCA til én dimensjon, ved å modifisere to linjer i koden over:

```
pca = PCA(n_components=1)
plt.scatter(X_pca, np.zeros_like(X_pca), color='red', label='PCA Transformed Data')
```

Dette gir oss plottet i figur 22. Merk at dette plottet strengt tatt ikke har en y-akse, da dataene er éndimensjonale. Vi har altså laget én feature fra to features. I dette eksemplet, som kun er ment for å gi en følelse av hva PCA gjør, har vi brukt hele datasettet til å tilpasse modellen, i `X_pca = pca.fit_transform(X)`. Vi kunne altså *ikke* brukt dette resultatet til å evaluere modellen.

Det er ikke overraskende at PCA fungerer godt til å transformere dataene i figur 21a til én dimensjon, da dataene ligger nærmest på en rett linje. La oss se på et nytt eksempel, nemlig de S-formede dataene i figur 23a. Ved hjelp av følgende kode genererer vi dataene, og gjør en PCA-transformasjon fra de opprinnelige tre til to dimensjoner. Resultatet vises i figur 23b. I koden under holder vi oss til god maskinlæringspraksis, og kjører `.fit` på treningsdataene, mens vi gjør `.transform` på testdata.

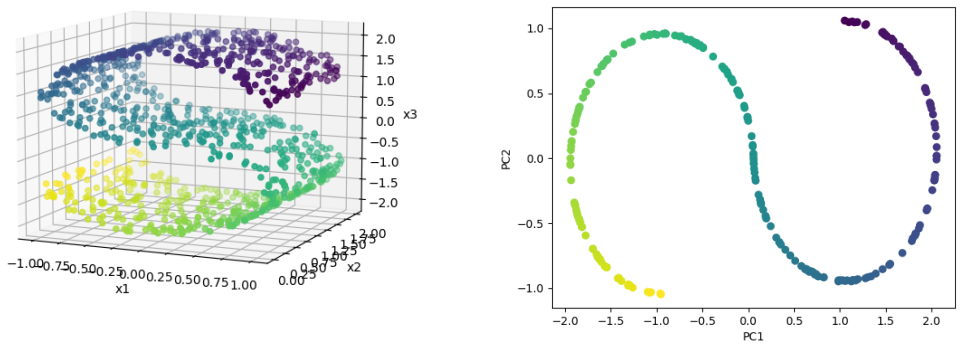
```
S_points, S_color = sklearn.datasets.make_s_curve(n_samples=1000)
x, y, z = S_points.T
pca = PCA(n_components=2)
X_train, X_test, y_train, y_test = train_test_split(S_points, S_color)
pca.fit(X_train)
S_pca = pca.transform(X_test)
plt.scatter(S_pca[:,0], S_pca[:,1], c=y_test)
plt.show()
```

Resultatet viser at PCA kan fungere på ikke-lineære data, så lenge variansen enkelt kan representeres langs uavhengige akser eller lineærkombinasjoner av disse – altså lineært. Ved å studere de S-formede dataene i figur 23a ser vi at dataene i dybderetningen har liten eller ingen varians, og det bør derfor ikke overraske oss at de kan representeres godt i to dimensjoner.

Før vi fortsetter skal vi analysere ett datasett til, nemlig MNIST-dataene som representerer håndskrevne tall fra 0 til 9, som vi så på tidligere i konteksten nevrale nettverk, se eksempler på datapunktene fra dette datasettet i figur 24a. Vi kan kjøre en PCA-analyse på disse dataene ved bruk av koden under, og resultatet vises i figur 24b. Her svarer fargene til labels `y_test`.

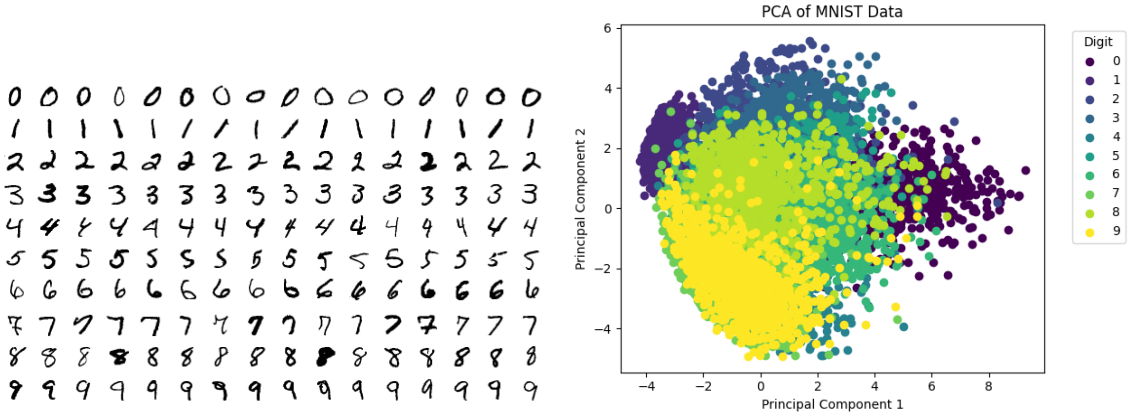
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255
y_train = y_train.astype("float32")
y_test = y_test.astype("float32")

pca = PCA(n_components=2)
pca.fit(x_train)
```



(a) (b)

Figure 23: (a) S-formede data i tre dimensjoner, og (b) hovedkomponenter PC1 og PC2 beregnet ved hjelp av PCA.



(a) (b)

Figure 24: (a) Eksempler på datapunkter fra MNIST-datasettet, og (b) PCA-transformasjon av MNIST-dataene fra 728 dimensjoner til to hovedkomponenter

```
X_pca = pca.fit_transform(x_test)
```

Som vi ser i figur 24b, er PCA-dekomposisjonen av MNIST-dataene fra 728 til to dimensjoner ikke egnet til å representere variansen i dataene særlig godt, da denne ikke kan beskrives som lineærkombinasjoner av uavhengige egenskaper. Vi skal derfor se på en annen metode.

3.2.2 Stochastic Neighbor Embedding (SNE)

Første steg i denne algoritmen er å tilpasse en sannsynlighetsfordeling som beskriver hvor like nabopunkter er. Likheten er i dette tilfellet en *betinget sannsynlighet*. For to datapunkter x_i og x_j kan vi beregne den betingede sannsynligheten $p_{j|i}$ for at x_j ville valgt x_i som nabo. For å beregne denne sannsynligheten bruker vi en Gaussisk fordeling $P(x)$, sentrert over datapunkt x_i, altså med forventningsverdi lik koordinatene til x_i. Da er det kun én parameter igjen i den gaussiske fordelingen, nemlig variansen. Det første SNE må gjøre er å tilpasse sannsynlighetsfordelingene, slik at punkter som ligger nærme hverandre har høy sannsynlighet for å være naboer, mens punkter som ligger langt fra hverandre har lav verdi. Dette gjøres i det opprinnelige datarommet, og når det er gjort, kan dimensjonsreduksjonen begynne. Prosessen er som følger:

1. Alle punktene flyttes til en lavere dimensjon, og plasseres tilfeldig.

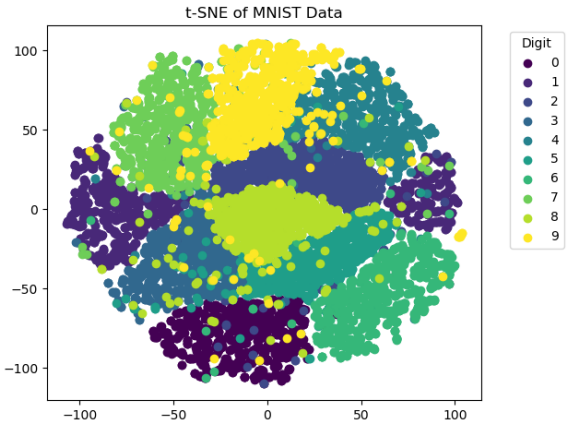


Figure 25: Resultatet av en t-SNE transformasjon av MNIST-datasettet.

2. Punktene flyttes slik at sannsynligheten mellom nabopunkter i den høye dimensjonen, beskrevet av fordeligen $P(x)$, matcher best mulig til den tilsvarende sannsynligheten i den nye dimensjonen, $Q(x)$.

Del 2 gjøres stegvis ved bruk av gradient descent, med tapsfunksjonen Kullback-Leibler (KL) divergence:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}. \tag{91}$$

KL-divergensen, også kjent som relativ entropi, måler likheten mellom en referansesannsynlighetsfordeling P og en annen sannsynlighetsfordeling Q. Merk at den ikke er symmetrisk i P og Q, og at den er strengt ikke-negativ. Legg gjerne merke til at den representerer forventningsverdien av den logaritmiske forskjellen mellom fordelingene P og Q, hvor selve forventningsverdien (summen) beregnes med datapunkter trukket fra P. **Oppgave:** Velg to ulike sannsynlighetsfordelinger P og Q og beregn KL-divergensen mellom de to.

En intuitiv tolkning av KL-divergensen er at den sier hvor overraskende resultater man vil få om man bruker sannsynlighetsfordeligen Q i stedet for P, når dataene man jobber med faktisk stammer fra P.

Fordelingen P måler i hvor stor grad punkter er naboer i det høydimensjonale rommet, mens Q måler det samme i det laveredimensjonale rommet. SNE minimerer KL-divergensen mellom fordelingene P og Q, altså: Sannsynligheten mellom nabopunkter i den høye dimensjonen, $P(x)$, matcher best mulig den tilsvarende sannsynligheten i den nye dimensjonen, $Q(x)$. Denne overføringen fra en høy- til en lavdimensjonal plassering av datapunkter kalles en embedding.

Til sammen beskriver dette prosedyren bak SNE. Denne ble imidlertid forbedret i 2008 av Laurens van der Maaten og Geoffrey Hinton, ved hjelp av en liten modifisering: i stedet for å bruke en Gauss i både den høye og lave dimensjonen, brukes en Gauss i den høye dimensjonen, og en t-fordeling i den lave. t-fordelingen likner på den gaussiske, men den faller litt raskere og har en lengre hale, hvilket minker sjansen for at nabopunkter får for like sannsynligheter. SNE som bruker t-fordelingen i den lavere dimensjonen, her fordelingen Q, kalles t-SNE. Denne er tilgjengelig i `sklearn`, og vi kan bruke den i stedet for PCA på MNIST-dataene ved bruk av følgende kommandoer.

```
tsne = TSNE(n_components=2, learning_rate='auto', init='random', perplexity=3)
X_embedded = tsne.fit_transform(X)
```

Resultatet vises i figur 25. Her har vi valgt verdien til en parameter `perplexity`. Denne er viktig for tilpasningen av variansen til Gaussen in det høydimensjonale rommet: perpleksiteten angir entropien til den betingede gaussiske fordelingen, og algoritmen tilpasser variansen for å matche denne. Denne er direkte relatert til antallet nære naboer vi forventer rundt et punkt. **Oppgave:** Se nøye på kodesnutten. Er det noe rart her?

I denne kodesnutten både tilpasses og brukes modellen `tsne` på de samme dataene. Dette er ikke i

tråd med god maskinlæringspraksis. Likevel har vi ikke noe valg, fordi `TSNE` er en *ikke-parametrisk* algoritme: den lager ikke en parametrisk modell vi kan ta med oss og bruke på nye data etter at den har blitt tilpasset. **Oppgave:** Klarer du å tenke deg frem til hvorfor en `TSNE`-modell ikke kan gjenbrukes?

`TSNE` optimaliserer posisjonen til datapunkter i det lavdimensjonale rommet, basert på parvise forhold i datasettet, altså sannsynligheter for naboskap med de andre punktene. Den lavdimensjonale embeddingen er avhengig av de relative avstandene mellom alle punktene i datasettet. Om vi legger til nye punkter, vil de relative avstandene og forholdene mellom punktene i datasettet endre seg. Da måtte hele embeddingen blitt beregnet på nytt for datasettet med de nye punktene.

PCA og t-SNE sammenliknet

- **Mål:** PCA Bevarer variansen i dataene; t-SNE minimerer divergensen mellom to sannsynlighetsfordelinger av dataene.
- **Gjenbrukbarhet:** PCA lager nye basisvektorer, og kan derfor brukes igjen etter `.fit`; t-SNE er ikke-parametrisk, må tilpasses og brukes samtidig, og kan derfor ikke gjenbrukes.
- **Feature-generering:** PCA kan brukes til å lage nye features, tilsvarende eller basert på hovedkomponentene; t-SNE kan ikke brukes til å lage nye features, siden den ikke lager en matematisk funksjon som kan brukes på nye datapunkter.
- **Linearitet:** PCA trenger lineært separerbare data; t-SNE har ingen antakelse om linearitet.
- **Compute:** PCA er rask og lite beregningstung, mens t-SNE må tilpasse mange sannsynlighetsfordelinger og derfor er treg på store datasett.

Oppgave: Hvilken av metodene tror du fungerer best på outliers, og hvorfor?

3.3 Anomalideteksjon

Når vi gjør anomalideteksjon, kjent som outlier detection eller anomaly detection, finner vi datapunktene som ikke likner majoriteten av datapunkter. Dette er relevant i mange sammenhenger, for eksempel antihvitvasking, kredittkortmisbuk, serverangrep, generelt tilfeller hvor vi vet hvordan normale datapunkt ser ut, men ikke nøyaktig hvordan avviket vil se ut. Vi har altså mange eksempler på normale datapunkter, men få eller ingen eksempler på anomalier. Dette kan tolkes som et klassifiseringsproblem med ekstrem skjevfordeling mellom klassene. Men siden vi oftest ikke vet hvordan den underrepresenterte klassen ser ut, er det vanligst å behandle anomalideteksjon som en uveiledet maskinlæringsoppgave.

I forelesningene ser vi nærmere på tre metoder, men som alltid kan dere bruke andre metoder i øvingene *så lenge dere skjønner grovt sett hvordan de fungerer*. Av de tre metodene vi vil studere kjenner vi allerede til to, nemlig k-means og DBSCAN, mens den siste, *isolation forest*, er ny.

3.3.1 k-means

Vi starter med å tilpasse en `kmeans`-modell tilsvarende som for clustering, på et treningsdatasett, med passende valg av antall sentroider k. Deretter beregner vi avstanden mellom testdatapunktene og nærmeste klyngesenter ved hjelp av

```
distances = kmeans.transform(X_test).min(axis=1)
```

Deretter må vi velge hvilke data vi vil se på som anomalier, og en mulighet er å bruke *prosentiler* (percentiles). Dette er verdien en viss prosentandel av datapunkter er mindre enn eller lik. Den 95. prosentilen representerer for eksempel datapunktene som har avstandsverdier større enn 95% av dataene. Vi gjør dette gjennom

```
threshold = np.percentile(distances, 95)
outliers = distances > threshold
```

Denne koden returnerer et array med `True` eller `False` per datapunkt, hvor `True` representerer datapunktene som har distance-verdi større enn 95% av dataene. Avhengig av dataene kan den 95.

persentilen inneholde for mange eller for få datapunkter, så det er godt mulig at terskelen må justeres. Det finnes ingen sann verdi for hvilken terskel som fungerer generelt.

3.3.2 DBSCAN

Vi starter med å tilpasse en `DBSCAN`-modell tilsvarende som for clustering. Her kan vi imidlertid ikke lage separate trenings- og testdata for `.fit` og `.transform`, siden DBSCAN er en ikke-parametrisk metode. Vi trenger altså tilgang til alle dataene på én gang. Dessuten må vi velge verdier for parametrene ϵ og `min_samples`, før vi kan hente ut outliers direkte ved hjelp av følgende kommando.

```
outliers = dbscan.labels_ == -1
```

Vi slipper altså å lage en regel for hvilke data vi vil regne som outliers, fordi DBSCAN er laget for å identifisere outliers.

3.3.3 Isolation Forest

En isolation forest er en skog, det vil si en samling besutningstrær, som identifiserer *isolerte* datapunkter, altså datapunkter uten mange naboer. Denne læringsalgoritmen er i utstrakt bruk for anomalideteksjon. Utfordringen er hvordan vi skal gå frem for å trene beslutningstrær uten å ha tilgang til labels.

Prosedyren består av følgende to steg: Først trenes flere *isolasjonstrær* – av *isolation tree*, forkortet itree – på et subsett av treningsdataene, trukket tilfeldig og uten tilbakelegging. Et isolasjonstre bygges ved å splitte på tilfeldige features og verdier av disse, inntil hvert datapunkt havner i en egen løvnode. Vi får altså én løvnode per datapunkt. Anomalier er få og ulike. Derfor havner disse som oftest i en node nærmere rotnoden, og har dermed kortere avstand fra rotnoden enn normale datapunkter. Dette gjentas for for mange isolasjonstrær, som gir en skog. Neste steg er å kombinere avstandene til rotnoden for hvert datapunkt. Denne kombinerte avstanden gir en god indikasjon på om et punkt er en anomali. Det er i prinsippet mulig å bygge kun ett isolasjonstre, men den tilfeldige trekningen av data for hvert isolasjonstre i skoen gir modellen robusthet til tross for tilfeldigheten i splittkriteriene. Vi kan bygge en isolation forest på treningsdata og bruke den til å finne outliers i testdata ved bruk av følgende kode.

```
iso_forest = IsolationForest(n_estimators=100, contamination=0.05)
iso_forest.fit(X_train)
outliers = iso_forest.predict(X_test) == -1
```

Modellprediksjonene med verdi -1 representerer anomalier, eller outliers. Som for alle metoder må vi tilpasse verdiene på hyperparametrene for å gå en god modell, og i koden over har vi valgt å bruke `n_estimators=100` isolasjonstrær, og anslått at dataasettet inneholder en andel `contamination=0.05` anomale datapunkter.

Oppsummering anomalideteksjon

Vi kan oppsummere fremgangsmåten for anomalideteksjon med de tre metodene som følger:

- **k-means**: Lager k klynger av dataene, basert på avstand mellom klyngesentrum og datapunktene. Anomalier er datapunkter som ligger langt unna alle klyngesentrene.
- **DBSCAN**: Lager klynger av punkter med høy tetthet, og identifiserer anomalier som isolerte punkter, altså outliers. DBSCAN finner outliers som en del av modelleringen.
- **Isolation Forest**: Bygger trær der alle punktene havner i en egen løvnode. Anomalier er punkter som har kort gjennomsnittlig avstand fra rotnoden, altså er lettere å isolere enn resten av datapunktene.

Oppgave: Hva er fordelene og ulempene med de ulike metodene?

3.4 Self-supervised learning

Dette avsnittet er basert på Ruslan Khalitovs forelesning 24.10.2024. Ruslan viser flere eksempler på modeller der self-supervised learning er aktuelt, men dere trenger kun å vite overordnet hva læringsformen går ut på. Eksemplene på autoencoders, diffusjonsmodeller osv er kun for å gi dere perspektiv og inspirasjon.

Self-supervised learning (SSL) er en del av feltet unsupervised learning, altså modellering basert på data uten labels. I tilfellet SSL genereres labels direkte fra dataene. Fremgangsmåten er som oftest at en andel av dataene holdes igjen, og fungerer som labels. En modell trenes på å predikere disse, basert på de resterende dataene. Denne prosessen omtales ofte som *pre-training*, og oppgaven der modellen predikerer en andel av dataene kalles en *pretext task*. Dette er ikke nødvendigvis nyttig i seg selv, men ofte ser vi at modellen lærer egenskaper fra dataene som er nyttige for videre modellering på nye data.

Neste steg i modelleringen kan være at vi henter nye data, som likner på dataene vi gjorde pre-trening på, denne gangen med labels. Om vi bruker modellen fra pre-treningen til denne nye, veiledede treningsoppgaven, omtales denne som *fine-tuning*. Slik fine-tuning er en variant av *transfer learning*, som brukes om alle tilfeller hvor en maskinlæringsmodell er trent på én oppgave, og så trener videre på en annen oppgave. Den videre treningen gjøres med en lav læringsrate, for at modellen ikke skal miste informasjonen den har hentet ut fra den første treningen. Fine-tuning innebærer altså ikke store endringer i modellens parametre. Likevel kan modellen gjerne bygges om mellom pre-trening og fine-tuning, for eksempel gjennom at siste lag byttes ut fra et lag med et stort antall noder til et lag med to noder, egnet til binær klassifisering.

Det viser seg at pre-trente modeller ofte gjør det langt bedre på veiledede oppgaver enn modeller som ikke har gått gjennom pre-trening, særlig i tilfeller hvor vi har lite data med labels, men store mengder data uten labels. Dette er ofte tilfellet for språkdata: Det finnes enorme mengder ustrukturerte språkdata uten labels, og langt mindre språkdata som er egnet for veiledet læring. Vanlige eksempler på pre-trente modeller finner vi derfor blant de store språkmodellene, inkludert GPT-familien, hvor P står for nettopp pre-trained (mens G står for generative, og T står for transformer). GPT-modellene er utviklet gjennom tre faser:

1. Pre-trening: Språkmodellen fullfører ulike setningsstykker, for eksempel fylle inn riktig ord i setningen `how are - doing today`.
2. Fine-tuning: Språkmodellen gjør språklige oppgaver med labels tilgjengelig, for eksempel predikere riktig stemning i en filmanmeldelse.
3. Reinforcement learning with human feedback: Språkmodellen lærer ytterligere, basert på tilbakemeldinger fra mennesker.

Dagens vellykkede anvendelser av SSL krever store mengder data, særlig til pre-treningen, og følgelig store mengder beregningsressurser.

Det finnes per i dag ikke ett rammeverk eller én teori for SSL; det er et forholdsvis ungt læringsregime innen maskinlæring, og de fleste anvendelsesområdene er regnet som state of the art. Derfor finnes det ikke mange lærebøker som dekker SSL per nå, og det forventes ikke at dere skal gjennomføre SSL selv i øvingene. Til eksamen trenger dere kun å kunne forklare overordnet hva SSL går ut på, og hvorfor, hhv i hvilke tilfeller, det er nyttig.

4 Reinforcement learning

Teorien bak *reinforcement learning*, forkortet RL, er stor og rikholdig, og i dette faget vil vi kun se på en del av den. Det finnes mange læringsregimer, og vi vil studere ett av dem, nemlig *Q-learning*. Dere får nok informasjon til å kunne trene en modell i et miljø dere får tilgang til i øving 3, men husk at å bli god i RL krever langt mer tid og fordypning enn vi har mulighet til å investere i dette kurset.

4.1 Markov Decision Processes

Reinforcement learning er en type maskinlæring hvor en *agent* lærer å ta beslutninger i et miljø, omtalt som *environment*, ved å utføre handlinger, omtalt som *actions*, for å maksimere forventet belønning,

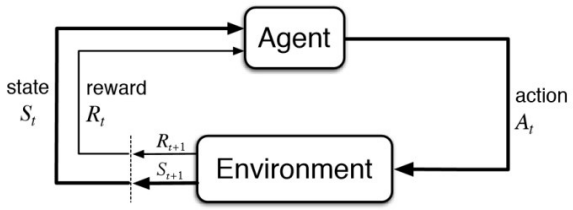


Figure 26: Reinforcement learning loop.

omtalt som *expected reward*. Denne prosessen går i en loop, se figur 26, som viser den såkalte RL-loopen. Her ser vi at environment (miljøet) gir agenten en state (tilstand) S_t ved steg t. Basert på state velger agenten en action (handling) A_t, som fører til en reward (belønning) R_{t+1} fra miljøet, og setter miljøet i en ny tilstand, state S_{t+1}. Basert på S_{t+1} velger agenten en ny action, og så videre i en loop. **Oppgave:** Ser du hvilken antakelse dette svarer til?

Agenten velger action basert kun på S_t, og får ikke informasjon om tidligere states. Dette svarer til en antakelse om at prediksjonen av den fremtidige tilstanden til environmentet kun er avhengig av inneværende tilstand og valgt action; ikke tidligere states eller actions. En slik prosess har *Markov property*, det vi si den egenskapen at den er uavhengighet av historien (hvordan agent og environment havnet i den aktuelle tilstanden). Slike prosesser kalles også memoryless. Kort oppsummert er Markov-prosesser slik at *fremtiden er uavhengig av fortiden gitt nåtiden*.

En verdenskjent maskinlæringsmodell som er trent på denne måten er AlphaZero, hvor 'zero' sikter til at modellen har lært kun gjennom å spille mot seg selv, uten tilgang til yttreligere informasjon. AlphaZero er et nevralt nettverk som får brettets tilstand som input state, og predikerer actions tilsvarende hvilke trekk som bør undersøkes nærmere i spilltreet. Spillets utfall til slutt gir reward på enten 1 om agenten vinner, -1 om den taper eller 0 for remis (uavgjort).

En beslutningsprosess som har Markov-egenskapen, kalles en *Markov decision process* (MDP). Formelt er en MDP en 4-tuple $(\mathcal{S}, \mathcal{A}, P_A, R_A)$, der

- $\mathcal{S}$ er set av states (state space),
- $\mathcal{A}$ er set av actions (action space). Alternativt: $\mathcal{A}_S$ er settet av actions tilgjengelige fra state S,
- $P_A(S_t, S_{t+1}) = P(S_{t+1}|S_t, A_t)$ er sannsynligheten for at action A_t i state S_t vil føre til state S_{t+1},
- $R_A(S_t, S_{t+1})$ er den umiddelbare rewarden for å transisjonere fra state S_t til state S_{t+1}, gjennom action A_t.

Oppsummert har vi en agent som skal ta beslutninger i et miljø. Problemet beskrives av states, actions, sannsynlighet for transisjon, og rewards. I dette kurset skal vi ikke se på transisjonssannsynligheter, som vil si at vi holder oss til deterministiske miljøer. Basert på en læringsprosess i miljøet som gir agenten states og rewards, skal agenten finne måte å velge ut beste action basert på en gitt state. Denne måten kalles en *policy*, og vi bruker notasjonen $\pi(A_t|S_t)$ for å angi en policy for å velge action basert på state. Målet til en RL-agent er (som regel) å finne en *optimal policy*, som maksimerer den totale rewarden, omtalt som kumulativ reward, fra miljøet.

Agenten kan være mye forskjellig, og vi skal først se på en tabell og deretter et nevralt nettverk som agent. For at denne skal kunne lære en optimal policy, må den trenes på å velge action som gir høy reward, og vi må ha en måte å beregne kumulativ reward. Dette gjør vi gjennom en *utility* eller *value* function. Denne angir hvor verdifull en tilstand og action er på lang sikt, under en viss policy. *Verdifunksjonen* kvantiserer *forventet verdi* av å være i en gitt tilstand, utføre en gitt action, følge en gitt policy, og fortsette å følge denne policyen fremover. Agenten trenger verdifunksjonen for å evaluere de langsiktige konsekvensene av en action og policy, i stedet for å bruke kun umiddelbar reward. **Oppgave:** Les de siste to setningene minimum tre ganger.

4.2 Q-verdier

Det finnes flere ulike verdifunksjoner, og i dette kurset studerer vi Q-funksjonen. Denne, hvor Q forkorter *q*uality, angir verdien (kvaliteten) til en state–action kombinasjon, som $Q : S \times A \to \mathbb{R}$. Vi beregner den ved bruk av Bellman-likningen:

$$Q^{new}(S_t, A_t) \leftarrow (1-\alpha) \cdot Q(S_t, A_t) + \alpha \cdot \left(R_{t+1} + \gamma \max_A Q(S_{t+1}, A) \right) . \tag{92}$$

Dette forteller oss at vi må beregne Q-verdien til et state-action-par iterativt. Prosessen består av følgende steg

1. Q initialiseres.
2. For hvert steg t har miljøet en tilstand S_t, agenten velger en action A_t, observerer en reward R_{t+1}, og får en ny tilstand S_{t+1}.
3. Q oppdateres basert på disse verdiene.

Vi ser på leddene i likningen over hver for seg:

- $Q^{new}(S_t, A_t)$ representerer den oppdaterte verdien av å være i S_t og gjøre A_t.
- $Q(S_t, A_t)$ er den gamle verdien av å være i S_t og gjøre A_t. Hvis man har en tabell for Q-verdiene er denne som oftest initialisert til 0, mens den initialiseres til en tilfeldig verdi hvis man har et nevralt nettverk som predikerer Q-verdiene.
- R_{t+1} er, som alltid, reward mottatt for å være i S_t og gjøre A_t,
- $\max_A Q(S_{t+1}, A)$ er maksimal verdi av state S_{t+1}, for alle tilgjengelige actions.

I tillegg har vi to parametre i likningen. Den ene, α, angir læringsraten, altså hvor mye av den gamle verdien av Q som beholdes, og hvor mye den nye verdien påvirkes av uttrykket i den siste parentesen. **Oppgave:** Hva skjer om $\alpha = 0$? Hva skjer om $\alpha = 1$? Den andre parameteren, γ, kalles *discount factor*, og gjør at rewards mottatt tidligere vektes ulikt fra rewards mottatt senere. **Oppgave:** Hva gjør en liten verdi av γ? Hva gjør en stor verdi av γ?

Forskjellen mellom verdi og reward er subtil, men viktig. Agenten får reward fra environmentet som en tilbakemelding på agentens actions i ulike states. Verdifunksjonen, i vårt tilfelle Q-verdiene, angir verdien av en state og action: den representerer den forventede verdien av å gjøre A_t i tilstand S_t, inkludert *fremtidig* verdi, under en gitt policy. Utover at reward og verdi representerer ulike ting, er det en springende forskjell at vi *får reward fra environmentet, mens Q må læres.*

Før vi ser på læring av Q-verdiene må vi forstå environments, og hvordan disse kodes og brukes.

4.3 Environments

Det er fult mulig å lage egne miljøer for reinforcement learning; alt de må kunne er å ha en tilstand, ta imot en action, transisjonere til en ny tilstand og returnere reward. Likevel er det vanligste, særlig når man lærer RL for første gang, å bruke tilgjengelige treningsmiljøer. Blant de mest brukte er OpenAI's `Gym`, og det er vanlig å starte med klassiske kontrollproblemer, inkludert `cartpole`, `mountaincar` og `pendulum` (se bilder i slides fra forelesningen). `Gym`-APIet har også gitt opphav til det nærmeste vi kommer en standardfunksjonalitet for environments brukt i RL. Spesifikt er det vanlig å ha en funksjon som returnerer environmentet til en utgangstilstand, og som returnerer denne tilstanden samt eventuell ekstra info:

```
env.reset() → Tuple[initial_state, info]
```

For å lage RL-loopen trengs i utgangspunktet kun én ytterligere funksjon, nemlig

```
env.step(action) → Tuple[state, reward, done, info]
```

Funksjonen `.step` tar altså en action som argument, hvorpå environmentet havner i en ny tilstand, som returneres sammen med eventuell reward mottatt for denne action, en boolean `done` som angir om environmentet har terminert, og eventuell ekstra info. Basert på de to kommandoene over kan man kode en RL-loop. Først settes environment i utgangstilstanden, og deretter lages en `for`-loop som

itererer over maks antall steg og terminerer når maks antall steg er nådd eller når `done==True`, eller en `while`-loop som terminerer når `done==True`. I denne loopen interagerer agenten med environmentet gjennom å sende inn actions, motta ny state og reward, og velge nye actions.

Som intuitivt eksempel kan vi se for oss spillet tre-på-rad. Starttilstanden `initial_state` er et brett med ni tomme ruter. Første spiller (menneske eller agent) plasserer en `'X'` på brettet, og plasseringen av denne representerer action. Dette gjør at environmentet inntar en ny state, med én rute okkupert av en `'X'`. Basert på denne staten plasserer den andre spilleren (menneske eller agent) en `'O'` på brettet, og environmentet inntar en ny state basert på denne. Slik fortsetter de to spillerne annenhver gang, frem til enten en av spillerne har fått tre av sine markører på rad (vertikalt, horisontalt eller diagonalt), eller til brettet ikke har flere tomme ruter igjen. I begge disse tilfellene settes `done=True`. Spilleren som får tre på rad vinner, og får `reward>0`. Hvis ingen får tre på rad er spillet uavgjort, og begge spillerne får `reward=0`.

Om vi vil skrive koden som lager dette environmentet, trenger vi minimum følgende funksjoner:

- `__init__`: Lager initial state, altså et tomt brett. Det kan for eksempel representeres som `'012345678'`. Setter `winner=None`, setter en variabel `player_turn='X'`. Kan eventuelt initialisere variabler som teller hvor ofte de ulike spillerne har vunnet, og andre ting som er nyttige å ta vare på.
- `reset`: Setter environmentet tilbake til initial state, `winner=None` og `player_turn='X'`, reinitialiserer eventuelle tellevariabler. Returnerer initial state og eventuelt en info-dictionary.
- `step`: Tar en action som argument og lager en ny state fra action, for eksempel `state='0123X5678'` hvis spiller `X` setter et kryss i midten. Sjekker om noen har vunnet, bytter på spillers tur (`X` hvis `O`, og omvendt). Returnerer ny state, reward, done og eventuelt info-dictionary.
- `check_winner`: Leter etter horisontale, vertikale eller diagonale linjer av `XXX` eller `OOO`. Returnerer `True` hvis det finnes en vinner, ellers `False`.
- `render`: Ikke nødvendig, men veldig nyttig om man vil se på environmentet under trening, eller når den ferdig trente agenten spiller. Lager en visuell representasjon av spillets tilstand.

Med et environment på plass – enten et vi har laget selv, eller et vi har importert fra et bibliotek – kan vi lage agenten som skal lære å løse en oppgave i environmentet.

4.4 Agenter: exploration, exploitation og læring

Oppgaven til en agent er å observere states og velge actions basert på disse. Vi holder oss til eksempelet med tre-på-rad, og ønsker å lage en agent som bruker Q-verdier til å velge actions basert på states. Da gjør vi såkalt *Q-learning*, og dette kan gjøres på flere måter. Vi vil først lage en enkel agent som har en tabell (dictionary) med Q-verdier, før vi ser på et nevralt nettverk som agent senere. Når vi initialiserer agenten må denne vite hvilken tag den har i spillet, altså `'X'` eller `'O'`, og den må ha et sted å lagre Q-verdier. Til dette bruker vi typisk en dictionary med state som key.

Se igjen på likning 92. For at agenten vår skal kunne lære, altså oppdatere Q-verdiene sine, må vi velge verdier for α og γ. Måten agenten vår får tilgang til treningsdata, er gjennom å utføre ulike actions i ulike states. For å sikre at agenten utforsker de tilgjengelige mulighetene i tilstrekkelig grad, brukes en hyperparameter vi ikke har vært borti før, og som kun gir mening i konteksten av reinforcement learning, nemlig `exploration_factor`, ofte angitt med symbolet epsilon ϵ. Denne styrer hvor mye agenten utforsker, på engelsk: *explore*, miljøet sitt. Den enkleste måten å få agenten til å utforske, er ved å velge tilfeldige actions med en viss sannsynlighet. Formelt velger vi en exploration factor $\epsilon \in [0, 1]$, som representerer sannsynligheten for at agenten gjør en tilfeldig handling. I funksjonen som velger action for agenten, skal denne velge action tilsvarende høyest Q-verdi, med sannsynlighet $1 - \epsilon$, og ellers en action tilfeldig samplet fra environmentets action space. **Oppgave:** Hva representerer henholdsvis en høy og en lav verdi av ϵ? Når i løpet av treningen er det lurt å velge tilfeldige actions? Bør agenten slutte å gjøre tilfeldige actions på et tidspunkt?

Hvis agenten utforsker (explore) miljøet, som svarer til en høy verdi av ϵ, finner den kanskje kombinasjoner av states og actions som fører til store belønninger. På den annen side kan agenten utnytte (exploit) det den vet om miljøet, som svarer til en lav verdi av ϵ, og unngå risikoen for å gjøre actions

som fører til store tap. Ulempen er at den da kanskje går glipp av belønninger dens aktuelle Q-verdier ikke vet om, fordi tilsvarende state-action-kombinasjon ikke har blitt besøkt før. Valget mellom utforsking og utnytting – som rommer faktumet at den ene går på bekostning av den andre, da de to svarer til henholdsvis høye og lave verdi av ϵ – kalles *exploration exploitation tradeoff*, og det finnes ingen universell løsning som gir en god tradeoff: Exploration handler om å ta actions med ukjent utfall, mens exploitation handler om å ta actions med kjent utfall. For å velge den optimale balansen mellom de to, måtte vi kjent utfallet til actions med ukjent utfall. Disse ville da per definisjon hatt kjent utfall, som er en selvmotsigelse. En mye brukt og ofte god strategi er *epsilon greedy*, med avtagende verdier av ϵ, såkalt *epsilon decay*. Da starter man med en høy verdi av ϵ tidlig i treningen, og reduserer denne stegvis. Dette fører til at agenten (nesten) slutter å gjøre tilfeldige actions sent i treningen, når den har lært seg mange av Q-verdiene. Når dette bør skje er avhengig av størrelsen på state-action-space, altså av environment og problemet som skal løses. Agenten slutter bare nesten å ta tilfeldige actions fordi det ofte fungerer bedre å beholde ϵ på en liten verdi, i stedet for å sette denne til nøyaktig 0, selv etter at treningen er avsluttet. Grunnen til det er simpelthen at selv en godt trent agent kan kjøre seg fast, og trenge et tilfeldig dytt for å komme seg videre.

Nå som vi har kontroll på de relevante parametrene, kan vi lage en klasse `Player`, hvor agenten er et objekt. Instanser av denne klassen som lærer å spille tre-på-rad må minimum ha følgende egenskaper

- tag $\leftarrow$ `'X'` eller `'O'`
- Q-tabell $\leftarrow$ en dictionary, inneholder alle Q-verdiene agenten kjenner, og må læres.
- $\alpha, \gamma, \epsilon \leftarrow$ agentens parametre. I tillegg kommer eventuelle læringsspesifikke parametre.

I forelesningene bruker vi tre-på-rad eksempelet for å lage agenter som lærer seg å spille optimalt for henholdsvis `X` og `O`. Koden samt ferdig trente agenter finnes på gitlab.com/Strumke/ntnu-lectures/-/tree/main/Tictactoe%20Q-learning.

I tillegg må agenten ha metoder for å velge action, lære, og oppdatere ϵ. Vi kan bruke følgende kodesnutt for å velge action.

```
def choose_action(self, state):
    available_actions = [_i for _i, _s in enumerate(state) if _s.isnumeric()]
    # If only one move available, make that one
    if len(available_actions) == 0:
        return None
    elif len(available_actions) == 1:
        action = available_actions[0]
        return action
    else:
        qvalues = []
        for _action in available_actions:
            potential_state = state[:_action] + self.tag + state[_action+1:]
            qvalues.append(self.model.predict(self.state_to_array(potential_state),
                verbose=0)[0][0])
        max_index = np.argmax(qvalues)
        best_action = available_actions[max_index]
        return best_action
```

Denne funksjonen gjør flere ting: 1) Får oversikt over mulige trekk. `state` er en string bestående av indeksene til mulige trekk og eksisterende plasseringer, for eksempel `0123XO678`. Mulige trekk er alle delene av denne stringen som er numeriske verdier. Hvis det ikke finnes tilgjengelige trekk, returnerer funksjonen `None`, og hvis det kun finnes ett tilgjengelig trekk, returneres dette. 2) Sjekker om `state` finnes i Q-tabellen. Hvis ikke settes staten inn med default Q-verdi 0. 3) Trekker et tilfeldig tall mellom 0 og 1. Hvis dette er mindre enn ϵ, returneres en tilfeldig action. 4) Hvis det tilfeldige tallet er større enn ϵ, returneres action tilsvarende høyeste Q-verdi for aktuelle `state`, i følge Q-tabellen.

4.5 Q-tabell

Q-læring handler om å oppdatere Q-verdiene etter hvert som agenten får erfaring fra environmentet, i vårt tilfelle ved å fylle dem inn i Q-tabellen gjennom læring. I starten av læringen vet agenten ingenting, og Q-tabellen er enten tom eller fylt av 0-er for alle states og actions. Som oftest programmeres en Q-tabell som en (nøstet) dictionary med states som keys. Dette kan nøstes slik at hver state igjen er en dictionary med actions som keys, slik at hver state har én entry per action, som igjen har en tilsvarende Q-verdi for det aktuelle state-action-paret. I starten av læringen spiller agenten som en komplett idiot, og gjør nærmest kun tilfeldige trekk. I slutten av hvert spill får agenten en reward, og denne brukes i læringsprosessen for å fylle inn stadig mer treffsikre verdier i Q-tabellen. Dette gjøres ved bruk av likning 92, gjentatt under for tilgjengelighet,

$$Q^{new}(S_t, A_t) \leftarrow (1-\alpha) \cdot Q(S_t, A_t) + \alpha \cdot \left(R_{t+1} + \gamma \max_A Q(S_{t+1}, A) \right) ,$$

for eksempel som i koden under.

```
def learn(self, state, action, reward, new_state, done):
    v_s = self.calc_value(state)[0]
    if done:
        v_s_next = 0
    else:
        v_s_next = self.calc_value(new_state)[0]
    td_target = reward + (0 if done else self.gamma*v_s_next)
    td_delta = td_target - v_s
    new_qvalue = v_s + self.alpha*td_delta
    y_train = new_qvalue
    model_input = self.state_to_array(state).reshape(1,-1)
    self.train_model(model_input, y_train, self.train_steps)
    return
```

Første linje i denne koden henter ut den gamle Q-verdien fra dictionariet. Deretter hentes den maksimale fremtidige verdien ut, med mindre den ikke eksisterer, i hvilket tilfelle den settes til 0. Deretter brukes disse to størrelsene samt parametrene med verdiene vi satte i starten av treningen, til å beregne den nye Q-verdien, som erstatter den gamle i dictionarien. Fuksjonen returnerer ingenting. Merk at læringen skjer for hvert trekk, og at funksjonen trenger å vite hvilken state agenten er i, hvilken action som ble valgt, hvilken state agenten havnet i, og hvor mye reward agenten mottok for trekket. I tilfellet tre-på-rad er `reward=0` med mindre spillet er over. Hvis spillet er over er maksimal mulig fremtidig verdi som regel 0 (ikke bare for tre-på-rad). Vi skjønner at *Q-læring går ut på å oppdatere Q-verdier gjennom erfaring.*

Til slutt trenger vi en funksjon som minker verdien av ϵ i løpet av treningen. Dette kan for eksempel løses ved hjelp av følgende funksjon.

```
def update_exp_factor(self):
    if self.exp_factor > self.exp_min:
        self.exp_factor *= self.epsilon_decay
```

Oppgave: Hva skjer her? Lag gjerne et plott av `exp_factor` over flere steg med ulike verdier av `epsilon_decay`. Det er en grei regel når man gjør RL at en agent som ikke har rukket å lære seg noe i løpet av perioden der `exp_factor>exp_min`, sannsynligvis ikke kommer til å lære noe vettugt om den trener videre. Da er det ofte mer hensiktsmessig å endre andre deler av koden, for eksempel tapsfunksjonen.

Koden som brukes i forelesningene – og finnes på github under lenken over – inneholder funksjonalitet for å plotte Q-verdiene for ulike states og actions underveis i spillet. En slik visualisering kan være nyttig for å forsikre seg om at verdiene gir mening. Det gir oss riktignok den samme informasjonen som om vi leser av Q-verdiene for actions gitt en state direkte, men gode visualiseringer kan likevel gjøre det enklere å ta informasjonen innover seg. Som eksempel kan vi laste inn Q-verdiene til en agent som har lært seg å spille tre-på-rad optimalt, og lese ut verdiene for tilstanden `state='012345678'`, altså det tomme brettet. Den aktuelle agenten (her vil numeriske variasjoner absolutt forekomme, så ikke se på de nøyaktige tallverdiene men heller størrelsesforholdene) har følgende Q-verdier.

```
In[4]: agent_x.values['012345678']
Out[4]:
{0: 1.9331107015552194,
 1: 1.5491666935002169,
 2: 1.5605398816666645,
 3: 2.139988821719487,
 4: 2.4760838915510397,
 5: 0.6561000000000463,
 6: 1.0237994971966147,
 7: 1.394265348233807,
 8: 1.6669174601240253}
```

Oppgave: Hva forteller disse Q-verdiene oss om hvor agenten vil plassere den første `X`-en på det tomme brettet?

I stedet for å lese av Q-verdiene numerisk, kan vi plotte dem på en grid tilsvarende spillbrettet. Merk: I det følgende eksempelet kan det være lurt å tegne opp et brett for å enklere kunne følge spillets utvikling. Vi starter med et tomt brett, og for det tomme brettet med `state='012345678'` vises agentens Q-verdier i figur 27a. Det er tydelig at agenten foretrekker å sette et kryss i midten av brettet, hvilket svarer til optimalt spill. Om vi setter en `O` slik at vi får neste `state='0123XO678'`, kan vi lese ut Q-verdiene vist i figur 27b. Ut ifra disse skjønner vi at agenten vil sette neste kryss på ruten nederst i midten. For å forhindre at agenten deretter får tre kryss på rad i midten, bør vi sette en `O` øverst i midten, slik at spillet ender opp i `state='0O23XO6X8'`. For denne tilstanden er agentens Q-verdier som vist i figur 27c. Vi skjønner at agenten vil sette et kryss neders i venstre hjørne, som gjør at vi må sette en `O` øverst i høyre hjørne for å forhindre at agenten får tre kryss på diagonalen. Dette setter spillet i `state='0OO3XOXX8'`, som igjen gir agenten Q-verdiene vist i figur 27d. Vi ser at agenten har skjønt at den får tre kryss på rad ved å sette et kryss nederst i høyre hjørne, og dermed vinner spillet. Gjennom denne analysen av Q-verdier får vi en intuisjon for hva de betyr: Agenten får ikke *reward* før spillet er over, men gjennom Q-verdiene kan vi forstå agentens *forventede fremtidige reward*. Sagt mer presist representerer Q-verdiene *forventet kumulativ fremtidig reward av å gjøre action A i state S og følge policyen deretter.*

4.6 Deep Q-learning

I stedet for å fylle (oppdaterte) Q-verdier inn i en dictionary, kan vi bruke et nevralt nettverk til å estimere Q-verdier, inkludert for tilstander agenten aldri har vært i. Når vi gjør reinforcement learning med dype nevrale nettverk, gjør vi *deep reinforcement learning*, forkortet DRL. I dette kurset holder vi oss til Q-læring, og når vi lager et dypt nevralt nettverk for å estimere Q-verdier, lager vi et såkalt *deep Q-network*, forkortet DQN.

En liten advarsel før vi går i gang: *DRL krever mye fikling å få til, og mye erfaring å beherske. Gjort riktig har DRL gjentatte ganger vist seg å gi opphav til nevrale nettverk med overmenneskelige problemløsningsevner og overraskende intelligent oppførsel, men veien til slike nevrale nettverk er lang. Dere trenger ikke gjøre DRL på noen av øvingene i dette emnet, og på eksamen vil kun konseptuelle spørsmål om DRL være relevante. Samtidig er DRL et tema som hos de fleste studenter krever en del modning, så om du ønsker å jobbe med DRL senere på masteren eller i livet, er det lurt å starte med temaet allerede nå.*

4.6.1 Environment: Frozen lake

Resultatene i forelesningen er produsert ved hjelp av koden i følgende repo: gitlab.com/Strumke/ntnu-lectures/-/tree/main/Frozenlake%20DQN. Her brukes `Gym`-environmentet `FrozenLake`, se figur 28. Vi kan laste inn environmentet og se på state- og action-space ved bruk av følgende kode.

```
import gym
env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=False)
#legg til render_mode='human' for visualisering
print(env.action_space)
print(env.observation_space)
```

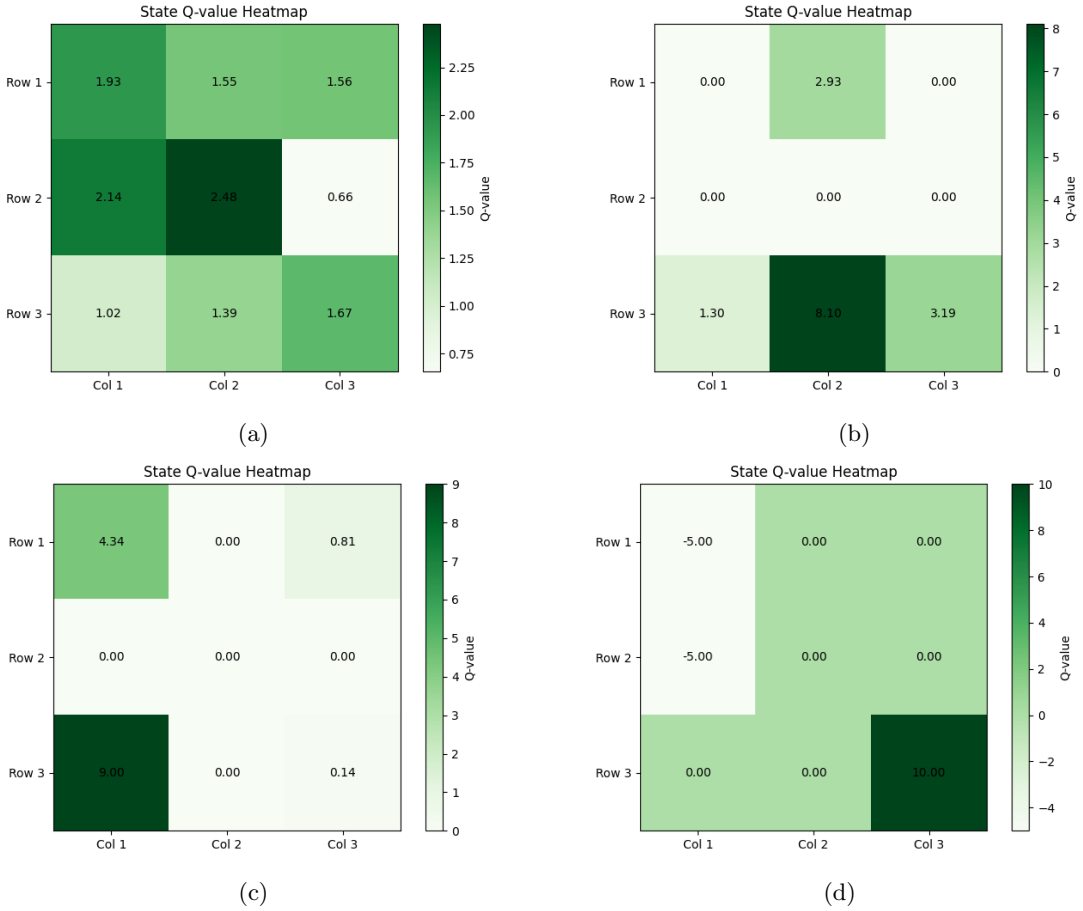


Figure 27: Q-verdier fra en agent med `tag='X'` som har lært å spille optimalt, for (a) et tomt brett, dvs `state='012345678'`, (b) brettet tilsvarende `state='0123X0678'`, (c) brettet tilsvarende `state='0023X06X8'`, og (d) brettet tilsvarende `state='0003X0XX8'`.

(a) (b)

Figure 28: Eksempler på visualiseringer av ulike states i Frozen Lake environment fra Gym.

Vi vil da se at action space er diskret med 4 tilgjengelige actions, nemlig 0 (move left), 1 (move down), 2 (move right) og 3 (move up). Videre ser vi at observation space også er diskret, med 16 mulige verdier, som tilsvarer at agenten kan befinne seg på hver av de 16 rutene. Som tidligere setter vi environmentet til utgangstilstanden gjennom `env.reset()`, og vi tar steg ved bruk av `.step` med valgt action som argument. Dette environmentet returnerer `done=False` inntil agenten faller i et av hullene, har gjort flere enn 100 steg, eller når julegaven som markerer mål. Hvis sistnevnte skjer returnerer environmentet `reward=1`, ellers er `reward=0`.

4.6.2 DQN: Q network og target network

Vi vil nå lage et nevralt nettverk som estimerer Q-verdier. Som alltid når vi lager nevrale nettverk, må vi starte med å finne riktig dimensjonalitet på input- og output-lagene. **Oppgave:** Hvor mange noder bør henholdsvis input- og output-laget ha i vårt tilfelle?

Prediksjonene fra nettverket skal være en Q-verdi per action, som betyr at vi trenger fire noder i output-laget. Agenten kan befinne seg på 16 ulike ruter, som i environmentet representerer gjennom diskrete tall fra 0 til 15. I stedet for å ha én input-node som tar inn et tall i dette invervallet, gjør vi lurt i å one-hot-encode input. Siden vi kommer til å lage det nevrale nettverket vårt i tensorflow, må input være en tensorflow-tensor. Da er det like greit å bruke tensorflow-funksjonalitet til å både lage og behandle tensoren, og vi kan preprosessere staten ved bruk av koden under.

```
def preprocess_state(state):
    map_size = 4
    num_states = map_size ** 2
    onehot_vector = tf.one_hot(state, num_states)
    return np.expand_dims(onehot_vector, axis=0)
```

Denne funksjonen tar inn en state representert av et tall i intervallet $[0, 15]$, og returnerer en tensor med shape $[1, 16]$, bestående av den one-hot-encodede versjonen av samme state. Dette representerer en input-batch med ett datapunkt. Nå som vi er enige om input- og output-dimensjonaliteten kan vi lage det nevrale nettverket, for eksempel ved bruk av koden under.

```
def create_model(self):
    num_actions = self.action_space.n
    input_dim = self.observation_space.n

    model = tf.keras.models.Sequential([
            tf.keras.Input(shape=(input_dim,)),
            tf.keras.layers.Dense(12, activation='relu'),
            tf.keras.layers.Dense(8, activation='relu'),
            tf.keras.layers.Dense(num_actions, activation='linear')
            ])
```

```
model.compile(optimizer='SGD', loss='mean_squared_error', metrics=['accuracy'])
model.summary()
print("Created model from scratch")
return model
```

Denne funksjonen bør tilhøre klassen agenten er en instans av. Hvis vi nå går hen og trener dette nevrale nettverket til å løse Frozen Lake, er sannsynligheten stor for at det kommer til å gå dårlig. For å skjønne hvorfor, må vi tenke på forskjellen mellom Q-læring og DQN. I Q-læring har vi en eksakt verdifunksjon, nemlig Q-funksjonen, mens en DQN er en *funksjonsapproksimator*, som skal estimere riktig Q-verdi for alle actions for hvilken som helst state. Som vi vet kan vi ikke endre én prdiksjon fra et nevralt nettverk – tilsvarende å oppdatere én verdi i Q-tabellen fra tidligere – men må oppdatere alle nettverksparametrene samtidig. Da endrer vi også Q-verdiene for actions i neste tilstand agenten befinner seg i - i stedet for at de er stabile som i Q-læring. Da oppstår følgende to problemer:

- Det nevrale nettverket lærer med et bevegelig mål.
- Det oppdaterte nettverket representerer en annen policy enn nettverket som predikerte forrige action. Dette strider mot antakelsen i Q-læring, om at man følger samme policy i fremtiden.

Den vanligste manifestasjonen av disse problemene er såkalt *catastrophic forgetting*, som vises gjennom at nettverket først lærer seg å løse oppgaven og mottar høy reward over en periode, før reward plutselig kollapser og nettverket begynner å gjøre dårlige prediksjoner igjen, selv om både læringsraten og ϵ har lave verdier. Denne syklusen vil fortsette, uansett hvor lenge nettverket får trene. Intuitivt skjer dette fordi nettverket lærer å løse problemet på nytt i stedet for å beholde representasjonen det genererte fra tidligere stadier i treningen; det glemmer i takt med at det lærer.

Problemene som forårsakes av at det samme nettverket representerer policy og oppdateres underveis, og at nettverket overtilpasser seg til nylige hendelser, kan løses ved hjelp av to virkemidler: Først innfører vi et nytt nevralt nettverk, ofte kalt `target network`, som utgjør et stabilt mål og en stabil policy under treningen. Dette nettverket er en kopi av vårt opprinnelige Q-nettverk, men det oppdateres med langt lavere frekvens. Vi bruker fremdeles det første Q-nettverket til å velge actions, men lar target network predikere $\max_A Q$ i Bellman-likningen under treningen av Q-nettverket. Intuitivt er target network en mer konservativ versjon av Q-nettverket. Under treningen setter vi en frekvens-parameter som styrer hvor ofte target-nettverket oppdateres, hvilket skjer gjennom at vi kopierer vektene fra Q-nettverket. I tillegg til target network har vi et ytterligere virkemiddel, nemlig *replay buffer*. I stedet for å oppdatere Q-nettverket etter hver action, som vi gjorde med verdiene i Q-tabellen, samler vi datapunkter i et treningsdatasett. Dette gjøres underveis i spillet, ved at states, actions, nye states, rewards og dones lagres i et minne, den såkalte replay buffer. Under treningen trekkes datapunkter fra denne bufferen, som regel tilfeldig, og denne trekningen av datapunkter utgjør en batch.

Nå er vi endelig klare for å instansiere klassen agenten vår skal bruke for å estimere Q-verdier, og det hele kan gjøres for eksempel ved hjelp av koden under.

```
class DQNAgent:
    def __init__(self, env, epsilon_max=0.999, epsilon_min=0.01,
            epsilon_decay=0.999, batch_size=32, update_frequency=10,
            learning_rate=0.001, discount=0.9, replay_size=4000,):

        self.epsilon_max = epsilon_max
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.discount = discount

        self.action_space = env.action_space
        self.observation_space = env.observation_space

        self.loss_func = tf.keras.losses.MeanSquaredError()
        self.optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)

        self.batch_size = batch_size
        self.replay_buffer_size = replay_size
        self.target_update_freq = update_frequency

        self.replay_buffer = ExperienceReplayBuffer(self.replay_buffer_size)

        # Q-value approximator
        self.modelfile = "model.keras"
        self.Q_network = self.load_model()
        # Stable target
        self.target_network = self.create_model()
        self.target_network.set_weights(self.Q_network.get_weights())
```

Siden vi allerede har lært hvordan epsilon decay gjøres, og at epsilon greedy går ut på å velge action tilsvarende høyeste Q-verdi med sannsynlighet $1 - \epsilon$, er det en smal sak å gjenbruke hhv tilpasse `update_epsilon` og `select_action` fra tidligere. Deretter er det kun én funksjon igjen vi trenger å finne ut hvordan vi skal lage, nemlig funksjonen som gjør selve læringen.

4.6.3 DRL: loss av Q-verdier og backpropagation

Funksjonen vi bruker for å la Q-nettverket lære må

- hente data fra replay buffer,
- estimere maksimal verdi for neste states ved hjelp av target network,
- bruke $\max_A(Q)$ til å beregne oppdatert Q-verdi,
- finne predikert Q-verdi for valgte actions (selv om disse ble valgt gjennom exploration) fra Q-nettverket,
- beregne loss mellom predikert Q-verdi og oppdatert Q-verdi, og
- backpropagate loss gjennom, altså trene, Q-nettverket.

Oppgave: Tenk over hvordan du ville gått frem for å gjøre det siste punktet på denne listen. Seriøst, tenk. Hent deg en kaffe, se ut vinduet og tenk nøye igjennom dette. Hva er utfordringen, hva er en mulig løsning? Hvis du ikke tenker igjennom dette går du potensielt glipp av en fantastisk læringsopplevelse, og har kun deg selv å takke.

Å trene Q-nettverket skjer gjennom at vi beregner loss på en batch, beregner gradienten til nettverksparametrene, og så oppdaterer parametrene basert på gradienten, ved hjelp av back propagation. Hittil har vi gjort dette indirekte gjennom funksjonen `model.fit(input, target)`. I vårt aktuelle tilfelle har vi dog ikke alle target-verdiene; vi har den oppdaterte Q-verdien for action agenten utførte, men ikke for de resterende actions, altså de resterende prediksjonene fra Q-nettverket. Vi har følgende,

- en Q-verdi per action fra Q-nettverket som skal trenes, og
- oppdatert Q-verdi (target) for den ene action agenten utførte.

Med dette har vi ikke det som trengs for å bruke `.fit()`. Det vi derimot kan gjøre, er å beregne loss mellom predikert og oppdatert Q-verdi for den action som ble valgt for å havne i neste state. Deretter kan vi bruke denne loss-verdien til å gjøre backpropagation, altså trene Q-nettverket. Følgende funksjon implementerer alt ovennevnte.

```
def learn(self):
    states, actions, next_states, rewards, dones = self.replay_buffer.sample(self.batch_size)
    states = tf.squeeze(states, axis=1)
    next_states = tf.squeeze(next_states, axis=1)

    next_target_q_value = tf.reduce_max(self.target_network(next_states), axis=1)
    # Use tf.where to set next_target_q_value to 0 where dones is True
    # (long version of next_target_q_value[dones] = 0)
    next_target_q_value = tf.where(dones,
            tf.zeros_like(next_target_q_value), next_target_q_value)

    y_train = rewards + (self.discount * next_target_q_value)
    # Wrap the forward pass and loss calculation in a tape block to record operations
    with tf.GradientTape() as tape:

        predicted_q = self.Q_network(states)
        # Gather the predicted Q-values corresponding to the selected actions
        batch_indices = tf.range(self.batch_size, dtype=tf.int64)
        indices = tf.stack([batch_indices, actions], axis=1)
        y_pred = tf.gather_nd(predicted_q, indices)

        loss = self.loss_func(y_train, y_pred)

    gradients = tape.gradient(loss, self.Q_network.trainable_variables)
    self.optimizer.apply_gradients(zip(gradients, self.Q_network.trainable_variables))
```

Denne funksjonen trekker data fra replay buffer, og forbereder tensorene `states` og `next_states` ved å gi dem riktig shapes. Deretter bruker den target network til å beregne $\max_A(Q)$, og bruker denne i Bellman-likningen til å beregne Q^{new}, altså i praksis targets `y_train`. Det neste som skjer, foregår inne i en `tr.GradientTape`-blokk. La oss først se hva som skjer inni blokken, før vi forstår hvorfor blokken er der. Inni blokken finnes Q-nettverkets predikerte Q-verdien for actions som ble gjort, og loss mellom disse og `y_train` beregnes. Etter blokken beregnes gradientene basert på loss, og disse brukes til å oppdatere nettverksparametrene ved bruk av optimizer, som vi tidligere valgte til å være SGD (stochastic gradient descent). Grunnen til at noen av disse operasjonene skjer inne i en blokk, er at tensorflow må ha oversikt over alle tensorene som er involvert i beregningen av størrelsene som skal brukes i backpropagation gjennom Q-nettverket. Derfor skjer prediksjonen fra Q-nettverket, operasjoner på tensorene og loss-beregningen i en blokk hvis hensikt er å be tensorflow følge med på alt som foregår i blokken. Merk at om vi hadde konvertert noen av tensorene for eksempel til numpy-tensorer, ville tensorflow mistet oversikten over dem, og gradientoperasjonen ville ikke fungert.

Med den nødvendige funksjonaliteten på plass, kan vi endelig gjøre RL-loopen. En minimal versjon av denne kan kodes som følger.

```
max_steps = 200
episodes = 3000
total_steps = 0

for _ep in range(episodes):
    state, _ = env.reset()
    state = preprocess_state(state)
```

```
done = False
truncated = False
for _step in range(max_steps):
    total_steps += 1
    action = agent.select_action(state)
    next_state, reward, done, truncated, _ = env.step(action)
    next_state = preprocess_state(next_state)
    agent.replay_buffer.store(state, action, next_state, reward, done)
    if len(agent.replay_buffer) > agent.batch_size:
        agent.learn()
        if total_steps % agent.target_update_freq == 0:
            agent.target_update()
    state = next_state
    if done or truncated:
        break
agent.update_epsilon()
```

Steg for steg gjør denne koden følgende: Maks antall steg per episode og totalt antall episoder fastsettes, og totalt antall steg initialiseres til null. Deretter går den ytre loopen over totalt antall episoder. Ved starten av hver episode settes environment tilbake til utgangstilstanden, og `done` og `truncated` settes til `False`. Den indre loopen går over maks antall steg per episode, og for hvert steg oppdateres `total_steps`. Nå kommer vi til delen av loopen(e) der magien skjer: action velges for aktuell state, og state, action, ny state, reward og done lagres i replay buffer. Hvis replay buffer har nådd en størrelse på minimum `batch_size`, trenes Q-nettverket. Hvis oppdateringsfrekvensen til target network er nådd, oppdateres også dette. Deretter settes ny inneværende state basert på forrige nye state, før episoden avbrytes hvis `done` eller `truncated` er `True`. Til slutt oppdateres ϵ etter hver episode.

4.6.4 Oversikt og resultat

Reinforcement learning blir fort uoversiktlig, siden vi ikke vet hvilke states agenten vil havne i og hvilke actions den vil velge. Kort sagt har vi ikke kontroll på treningsdataene som genereres i løpet av læringen, i tillegg til at treningen preges av et stort antall (hyper)parametre vi må finne gode verdier for. Derfor er det lurt å enten plotte utviklingen av informative størrelser for å kunne analysere dem i lys av hverandre, for eksempel verdien til ϵ, total reward per episode, og totalt antall steg brukt. Dette kan gjøres manuelt, eller man kan bruke et av mange gode verktøy utviklet for dette formålet. Mange synes at weights and biases er lettvindt og nyttig, og figur 29 viser verdien til ϵ, antall steg per episode og hvorvidt agenten nådde målet, per totalt antall steg. Her ser vi at ϵ minker som forventet, i takt med at agenten bruker først flere steg per episode, før antall steg per episode går ned og mer eller mindre stabiliserer seg på en lav verdi. Dette tyder på at agenten først går gjennom en periode der den stort sett faller i vannet, hvilket avbryter episoder, deretter entrer en periode der den har delvis lært å unngå vann men bruker mange steg på å enten finne mål eller falle i vannet igjen, før den til slutt lærer å bruke få steg på å nå mål. Denne analysen støttes av det faktum at plottet til høyre viser at agenten først nesten alltid når målet, før den når målet sporadisk, og til slutt når målet i nesten hver episode. Basert på disse plottene kan vi være ganske sikre på at agenten har lært en god policy for å nå målet uten å falle i vannet. Til slutt er det lurt å lage en loop hvor vi visuelt inspiserer agentens oppførsel, og koden under kan brukes til dette formålet.

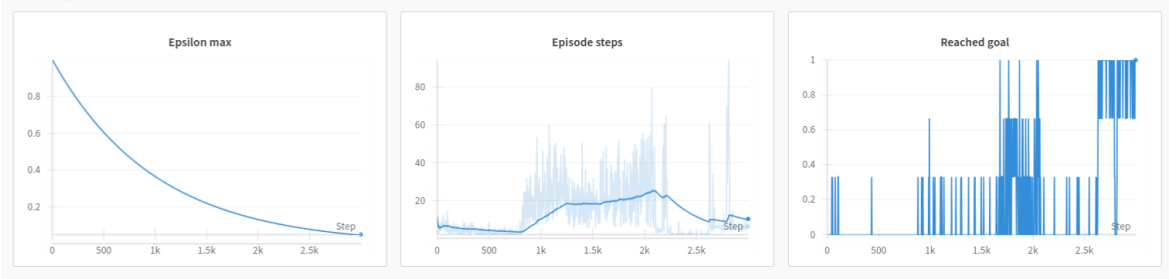


Figure 29: Plott fra weights and biases som viser ϵ, steg per episode og hvorvidt agenten nådde målet, per totalt antall steg.

```
env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=False, render_mode="human")
agent = dqn_agent.DQNAgent(env, epsilon_max=0.1)

done, truncated = False, False
state, _ = env.reset()
state = preprocess_state(state)
while not done:
    env.render()
    action = agent.select_action(state)
    print(state)
    print(action)
    state, reward, done, truncated, _ = env.step(action)
    state = preprocess_state(state)
    if done or truncated:
        break
```

Oppgave: Bli klok på hva som skjer i denne kodesnutten.

5 Eksempler på eksamensspørsmål

- Hva representerer diagonalen $(0,0) - (1,1)$ på en ROC-kurve?
  1. En klassifiseringsmodell med samme treffsikkerhet som tilfeldig gjetning.
  2. En klassifiseringsmodell som i gjennomsnitt predikerer 0.5 for alle dataene brukt til å lage ROC-kurven.
  3. En klassifiseringsterskel på 0.5.
  4. Ytelsen til en middels god klassifiseringsmodell, som vi bør ligge over for å ha laget en modell som yter godt.
- Hvilket av følgende er ikke en korrekt tolkning av ROC AUC?
  1. En høy ROC AUC indikerer god evne til å skille mellom klasser over ulike terskler.
  2. ROC AUC gir en sannsynlighet for at modellen vil rangere et positivt eksempel høyere enn et negativt eksempel.
  3. En lav ROC AUC betyr at modellen har lav presisjon.
  4. En ROC AUC på 0.5 betyr at modellen gjetter tilfeldig mellom klassene.
- Hvilken del av maskinlæringsprosessen er prediksjon?
  1. Iterativ beregning av riktig parameterverdi ved hjelp av en optimaliseringsprosess og den deriverte av tapsfunksjonen.
  2. Estimat av funksjonsverdi basert på data..
  3. Evaluering av utvalgte metrikker på usette testdata.
  4. Sammenlikning av metrikker på usette testdata, med verdiene tidligere beregnet på treningsdataene.
- Egenverdiene til følgende matrise
$$\begin{bmatrix} 5 & 0 \\ 5 & 1 \end{bmatrix} \tag{93}$$
er $\lambda = 5$ og $\lambda = 1$. Hva er egenvektorene?
  1. $(5,1)$ og $(1,0)$
  2. $(4/5,1)$ og $(0,1)$
  3. $(5/4,1)$ og $(0,1)$
  4. $(4/5,1)$ og $(1,0)$
- Hva er typiske konsekvenser av å bruke for liten batch size under trening?
  1. Det blir mer støy i gradientene
  2. Hver treningsepoke tar mindre tid
  3. Det blir mindre støy i gradientene
  4. Modellen overtilpasser
- Hva betyr det hvis KL-divergensen i t-SNE er veldig høy etter mange iterasjoner?
  1. Det betyr at dataene har høy varians og at t-SNE ikke klarer å finne en god lavdimensjonal projisering.
  2. Det betyr at den lavdimensjonale representasjonen av dataene ikke bevarer sannsynligheten for naboskap sammenliknet med den høydimensjonale representasjonen.
  3. En høy verdi betyr at t-SNE har funnet den optimale embeddingen fra den høydimensjonale til den lavdimensjonale representasjonen av dataene.

4. Det betyr at vi har valgt for lav perplexity-parameter i algoritmen.

- Hvilken av disse egenskapene må være oppfylt for at et datapunkt skal være en anomali i datasettet?
    1. Datapunktet er langt unna alle klyngesentre.
    2. Datapunktet kan plukkes ut ved hjelp av k-means clustering og persentiler.
    3. Datapunktet markeres som en outlier av DBSCAN.
- Hvordan påvirker Markov-egenskapen verdifunksjonen $V(s)$ i konteksten reinforcement learning?
    1. Verdifunksjonen $V(s)$ avhenger bare av belønningen i den nåværende tilstanden.
    2. Verdifunksjonen $V(s)$ avhenger av hele sekvensen av tidligere tilstander.
    3. Verdifunksjonen $V(s)$ avhenger av fremtidige belønninger basert på den nåværende tilstanden og handlingene som følger.
    4. Verdifunksjonen $V(s)$ er konstant for alle tilstander hvis Markov-egenskapen gjelder.
- Hva gir en global XAI-metode?
    1. En forklaring av hvorfor modellen gjorde en spesifikk prediksjon
    2. En overordnet forklaring på hvordan modellen tar beslutninger på tvers av datasett
    3. En forklaring på hvordan en hvilken som helst modell fungerer, uavhengig av modelltype
    4. En forklaring på modellens oppførsel samt et mål på hvor treffsikker forklaringen er.
- Hvilket av disse utsagnene er riktig om AI Act?
    1. AI Act ble vedtatt i 2018
    2. AI Act ble gjeldende i Norge i August 2024
    3. AI Act pålegger alle AI-systemer å være rettferdige
    4. AI Act gjelder alle som tilbyr AI-tjenester i EU