LINEAR MODELS - LINEAR REGRESSION

Linear regression er en linear model som forsøger, at finde en hyperplane i vore data, som fortæller os et tal givet en data vektor. Imodsætning til classification, hvor resultatet er binært, vil linear regression give os en $\mathbb{R}$ værdig. Dette kunne fx være hvor meget skal en bank låne en kunde. Antagelsen man skal gøre, for at bruge linear regression er, at der findes en linear kombination af informationer, som kan approximere, hvad vi gerne vil approximere.

Denne form for læring bruger squared error imellem $h(\mathbf{x})$ og y til at estimere E_{out}:

$$E_{out}(h) = \mathbb{E}\left[(h(\mathbf{x}) - y)^2\right]$$

Hvor den forventede værdi er taget ift. den forenede sandsynligheds fordeling $P(\mathbf{x}, y)$. Målet er selfølgelig at opnå en så lille $E_{out}(h)$ som muligt. Vi finde $E_{in}(h)$ ved:

$$E_{in}(h) = \frac{1}{N}\sum_{n=1}^{N}(h(\mathbf{x_n}) - y_n)^2$$

hvor

$$h(\mathbf{x}) = \sum_{i=0}^{d} w_i x_i = \mathbf{w}^T\mathbf{x}$$

hvor $x_0 = 1$ og $\in \{1\} \times \mathbb{R}^d$ og $\mathbf{w} \in \mathbb{R}^{d+1}$. Når vi har med et *lineært* h at gøre, er det meget brugbart at have en matrix repræsentation af $E_{in}(h)$. Definer $X \in \mathbb{R}^{N\times(d+1)}$, hvor hver row er et input $\mathbf{x}_n$ også har vi:

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - y_n)^2 = \frac{1}{N}||X\mathbf{w} - \mathbf{y}||^2 = \frac{1}{N}(\mathbf{w}^T X^T X\mathbf{w} - 2\mathbf{w}^T X^T\mathbf{y} + \mathbf{y}^T\mathbf{y})$$

Linear regression algoritmen fås ved at minimere $E_{in}(\mathbf{w})$, og alle mulige $\mathbf{w} \in \mathbb{R}^{d+1}$. Derfor er vi interesseret i følgende optimeserings problem:

$$\mathbf{w}_{lin} = \underset{\mathbf{w}\in\mathbb{R}^{d+1}}{\arg\min} E_{in}(\mathbf{w})$$

Siden $E_{in}(\mathbf{w})$ er differentiabel, kan vi finde dens gradient ved, og løse den for $\nabla E_{in}(\mathbf{w}) = \mathbf{0}$

$$\nabla E_{in}(\mathbf{w}) = \frac{2}{N}(X^T X\mathbf{w} - X^T\mathbf{y})$$

For at løse $\nabla E_{in}(\mathbf{w}) = \mathbf{0}$, finder vi et $\mathbf{w}$ der opfylder

$$X^T X\mathbf{w} = X^T\mathbf{y}$$

Hvis $X^T X$ er invertible, hvilket det er i de fleste tilfælde er, kan vi finde den unikke optilmale løsning for $\mathbf{w}$ ved

$$\mathbf{w} = (X^T X)^{-1}X^T\mathbf{y} = X^\dagger\mathbf{y}$$

som er vores $\mathbf{w}_{lin} = X^{\dagger}\mathbf{y}$. Denne giver vores hypotese som esitmere $\mathbf{y}$, som $\hat{\mathbf{y}} = X\mathbf{w}_{lin}$, som afviger fra $\mathbf{y}$ grundet in-sample errors.

Fører dette til en god E_{out}? Det korte svar er ja, hvor det gælder at

$$E_{out}(h) = E_{in}(h) + O\left(\frac{d}{N}\right)$$

LIENAR MODELS - PERCEPTRON LEARNING ALGORITHM

Perceptron er en learning algoritme der bruges til linear clasification af data, igennem iterationer hvor man opdatere vægtninge af forskellige d-dimensionel data, så nogle dimensioner er vigtigere end andre.

Vi bruger algoritmen til, at automatisere classification problemer, eller problemer hvor der skal afgøres noget med binære muligheder.

Vi specificere en hypothese $\mathscr{H}$, og finder vores hypothese $h \in \mathscr{H}$. Denne defineres som

$$h(x) = sign\left(\left(\Sigma_{i=1}^{d} w_i x_i\right) + b\right) = sign(\mathbf{w}^T\mathbf{x}) \in \{0,1\}$$

Igennem $t = 0,1,2,...$ iterationer, opdatere vi de forskellige weights for *misclassified data* ved.

$$\mathbf{w}(t+1) = \mathbf{w}(t) + y(t)\mathbf{x}(t)$$

Værdierne for $\mathbf{w}$ er initialiseret til at være forskellige *tilfældige* værdiger. Vi har at $w_0 = b$ og $\mathbf{w} = [w_0, w_1, ..., w_d]^T \in \mathbb{R}^{d+1}$ og vores space for x er

$$\mathscr{X} = \{1\} \times \mathbb{R}^d = \{[x_0, x_1, ..., x_d]^T | x_0 = 1, x_1, ..., x_d \in \mathbb{R}\}$$

Efter t itertioner vil PLA stoppe og vi har at $E_{in}(\mathbf{w}_{PLA}) = 0$, *hvis* dataen er linear seperable! Hvis dataen ikke er linear seprable skal vi approximere en løsning, fx igennem pocket algorithm, som for hver gang vi finder en$E_{in}(\hat{\mathbf{w}}) < E_{in}(\mathbf{w})$, gemmer $\hat{\mathbf{w}}$, og iterere videre, hvor den efter et valgt t iterationer stopper og returnerer den bedst observerede $\mathbf{w}$. Det man giver afkald på med pocket algorithm er ift PLA som opdatere værdier i $\mathbf{w}$ og checker for nogle exempler, de misclassified, bliver pocket algorithm nødt til, at checke for alle, for at kunne berenge $E_{in}(\mathbf{w})$.

Det smarte ved PLA, er at den iterere igennem et uendeligt stort hypothese space, i (beviseligt) endelig tid.

LINEAR MODELS - LOGISTIC REGRESSION

Logistic regression er en linear model som forsøger at fortælle os sandsynligheden for et given indput finder sted. Dette kunne være sandsynligheden for en patient får et hjertestop, givet hans journal. Grunden til denne form for model er, imodsætningen til linear classification som har en *hard threshold* og linear regression som har *no threshold*, så har logistic regression et *soft threshold*, mellem $[0-1]$.

Vi definere vores hypothese h i linear regression således

$$h(\mathbf{x}) = \theta(\mathbf{w}^T\mathbf{x})$$

hvor θ er en såkaldt logistisk function, fx $\theta(s) = \dfrac{e^s}{1+e^s}$ eller $\theta(s) = tanh(s) = \dfrac{e^s - e^{-s}}{e^s + e^{-s}}$.

Vores target function vi prøver at ville lære er defineret som $f(\mathbf{x}) = P[y = 1|\mathbf{x}]$, men det vi kigger på er en mængde data, hvor vi kender udfaldet, som er generet af et *noisy target* $P(y|\mathbf{x})$

$$P(y|\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{for } y = +1 \\ 1 - f(\mathbf{x}) & \text{for } y = -1 \end{cases}$$

Derfor må vi definere en *error measure* som måler hvor tæt hypothese h er på f, ift vores *noisy* ± 1 eksempler. Måle formen for fejl er $e(h(\mathbf{x}), y)$ som er baseret på *likelihood*, altså hvor sandsynligt y er givet $\mathbf{x}$. Vi kan sunstituere $h(\mathbf{x})$ til $\theta(\mathbf{w}^T\mathbf{w})$ og bruger det faktum at $1 - \theta(s) = \theta(-s)$ og får

$$P(y|\mathbf{x}) = \theta(y\mathbf{w}^T\mathbf{w})$$

Vi definere på samme tid vores $E_{in}(\mathbf{w})$ til

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n\mathbf{w}^T\mathbf{x}_n}\right)$$

Dette betyder vores *pointwise error measure* er $e(h(\mathbf{x}_n), y_n) = \ln(1 + e^{-y_n\mathbf{w}^T\mathbf{x}_n})$, hvor vi kan se at vores *error measure* er lille når $y_n\mathbf{w}^T\mathbf{x}_n$ er stor og *positiv*, hvilket ville betyde at $sign(\mathbf{w}^T\mathbf{x}_n) = y_n$

For at træne logistisk regression vil vi forsøge at sette $\nabla E_{in}(\mathbf{w}) = \mathbf{0}$, dette er dog et *svært* problem. Derfor løses det med en iterativ algoritme *gradient descent*. En stor fordel ved denne tilgang er, grundet vores logistiske regression benytter cross-entropy error, at den function vi skal descente ned af convex, og vi vil derfor altid gå mod et globalt minimum. Vi øsnker at descente ned af, den stejleste vej, for at nå vores mål hurtigts. Så vi definere η til at være vores step-size, og $\hat{\mathbf{v}}$ til at være en enhedsvektor. Vores nye wieght er da $\mathbf{w}(0) + \eta\hat{\mathbf{v}}$ Det er dog ikke en god idé at fasthold η igennem alle iterationer, da det giver god mening at tage mindre skridt, jo tættere vi kommer på det globale minimum, så denne kan justeres hen af vejen.

Dette fører os til algoritmen for logistik regression

1. initialiser weigths ved tid $t = 0$ til $\mathbf{w}(0)$
2. **for** $t = 0, 1, 2, ...$**do**
   a) beregn gradienten

$$\mathbf{g}_t = -\frac{1}{N}\sum_{n=1}^{N}\frac{y_n\mathbf{x}_n}{1+e^{y_n\mathbf{w}^T(t)\mathbf{x}_n}}$$

   b) sæt bevægelses retning, $\mathbf{v}_t = -\mathbf{g}_t$
   c) updater weigths: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta\mathbf{v}_t$
   d) iterer til næste step intil tiden stopper
3. Returner final weigth $\mathbf{w}$.

Ved kørsel af algortimen sættet $\mathbf{w}(0)$ til tilfældige tal, for at undgå vi sidder fast på en symmetrisk top, og vi vælger selv tid t for hvornår algoritmen skal stopper. Dette kan være når $||\mathbf{g}_t||$ er mindre end et bestemt kriterium, eller blot efter en arbitrær tid t.

En anden udgave af gradient descent delen af vores algoritme er stochastic gradient descent, hvor der vælges et tilfældigt punkt, som vi opdatere vores $\mathbf{w}$ ud fra, is tedet for at gøre det på hele data mængden, hvilket gør den af algoritmen hurtigere.

LEARNING THEORY - VC DIMENSION

Vi introducere ideen om *growth function*, og til det skal vi bruge dichotomies

DEFINITION 2.1. Lad $\mathbf{x}_1, ..., \mathbf{x}_N \in \mathscr{X}$. Dichotomies generaret af $\mathscr{H}$ for disse points er defineret som :

$$\mathscr{H}(\mathbf{x}_1, ..., \mathbf{x}_N) = \{(h(\mathbf{x}_1), ..., h(\mathbf{x}_N)) | h \in \mathscr{H}\}$$

disse kan ses et par briller hvor igennem vi kigge på hele $\mathscr{H}$ rummet igennem N punkter og kan se forskel på to h hvis de er forskellige ift deres vurdering af $\mathbf{x}_1, ..., \mathbf{x}_N$. En growth function er defineret på antallet af dichotomies

DEFINITION 2.2. Growth function er defineret for en hypothese mængde $\mathscr{H}$ ved

$$m_{\mathscr{H}} = \max_{\mathbf{x}_1, ..., \mathbf{x}_N \in \mathscr{X}} |\mathscr{H}(\mathbf{x}_1, ..., \mathbf{x}_N)|$$

altså er $m_{\mathscr{H}}$ det største antal dichotomies der kan generes af $\mathscr{H}$ for alle N points. Hvis $\mathscr{H}(\mathbf{x}_1, ..., \mathbf{x}_N) = \{0,1\}^N \implies m_{\mathscr{H}}(N) = 2^N$, og vi siger at $\mathscr{H}$ kan *shatter* $\mathbf{x}_1, ..., \mathbf{x}_N$.

DEFINITION 2.3. Hvis intet dataset af størrelse k kan *be shattered* af $\mathscr{H}$, så er k et *breakpoint* for $\mathscr{H}$.

Hvis definition 2.3 gælder, så ser vi at $m_{\mathscr{H}}(k) < 2^k$.

THEOREM 2.4 Hvis $m_{\parallel} < 2^k$ for en værdi k, så

$$m_{\mathscr{H}}(N \leq) \sum_{i=0}^{k-1} \binom{N}{i}$$

for alle N.

hvilket fører os til

DEFINITION 2.5 Vapnik-Chervonenkis dimensionen af en hypothese mængde $\mathscr{H}$, noteret ved $d_{VC}(\mathscr{H})$ eller d_{VC}, er den største værdi N, der gælder at $m_{\mathscr{H}}(N) = 2^N$. Hvis $m_{\mathscr{H}}(N) = 2^N$ for alle N, så er $d_{VC}(\mathscr{H}) = \infty$.

Grunden til at VC dimensioner er vigtige er, både at de angiver hvor mange *degrees of freedom* vi har at arbejde med i vores hyperplane men også, at hvis vi ikke kan finde et data set der kan shatter $m_{\mathscr{H}}$ har vi

THEOREM 2.5 (VC GENERATLIZATION BOUND). For er vilkårlig tolerance $\delta > 0$,

$$E_{out}(g) \leq E_{in}(g) + \sqrt{\frac{8}{N} \ln \frac{4m_{\mathcal{H}}(2N)}{\delta}}$$

Med sandsynlighed $\geq 1 - \delta$. Vi definere også $\sqrt{\frac{8}{N} \ln \frac{4m_{\mathcal{H}}(2N)}{\delta}} = \omega(N, \mathcal{H}, \delta)$, som værende modellens complexitet, altså som en slags penalty for hvor complex vores model er, og som straffer os med ringere E_{out}.

Vi ser hurtigt at hvis $m_{\mathcal{H}} = \infty$ har vi ingen garanti for at vi kan lære noget givet.

Vi har altså fået givet med VC analyses, at vi skal vores valg af $\mathcal{H}$ skal finde en balance imellem at approximere f på trænings daten, og generalisere over nyt data.

Lad os definere $\bar{g} \approx \frac{1}{K}\sum_{k=1}^{K} g_k(\mathbf{x})$ for alle $\mathbf{x}$. Givet out of sample error på squared error, har vi

$$\begin{aligned}\mathbb{E}_{\mathscr{D}}[E_{out}(g^{(\mathscr{D})})] &= \mathbb{E}_{\mathbf{x}}[\mathbb{E}_{\mathscr{D}}[g^{(\mathscr{D})}(x)^2] - 2\bar{g}(\mathbf{x})f(\mathbf{x}) + f(\mathbf{x})^2] \\ &= \mathbb{E}_{\mathbf{x}}[\underbrace{\mathbb{E}_{\mathscr{D}}[g^{(\mathscr{D})}(\mathbf{x})^2] - \bar{g}(\mathbf{x})^2}_{\mathbb{E}_{\mathscr{D}}[(g^{(\mathscr{D})} - \bar{g}(\mathbf{x}))^2]} + \underbrace{\bar{g}(\mathbf{x})^2 - 2\bar{g}(\mathbf{x})f(\mathbf{x}) + f(\mathbf{x})^2}_{(\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2}]\end{aligned}$$

Det sidste term kalder vi for bias

$$\text{bias}(\mathbf{x}) = (\bar{g}(\mathbf{x}) - f(\mathbf{x}))^2$$

Bias er et mål for hvor *skæv* vores læringsmodel ift vores target function. Hvor godt kan vi rent faktiske *fit* vores data, i gennesnit(fejl ift idel h). På samme måde definere vi

$$\text{var}(\mathbf{x}) = \mathbb{E}_{\mathscr{D}}[(g^{(\mathscr{D})}(x) - \bar{g}(\mathbf{x}))^2]$$

Variance måler hvor meget vores endelige hypothese variere ift vores data set. Hvor meget vi data i gennemsnit lede mig væk fra f.

Lærings algortimen spiller en roller ift bias-variance analyse som den ikke gør med fx VC analyse. Her er der specielt to ting at vore opmærksom på

- I modsætning til VC analys som er baseret kun på hypthese mængden $\mathscr{H}$, uafhængigt af lærings algoritmen $\mathscr{A}$, er bias variance analyse baseret på både $\mathscr{H}$ og $\mathscr{A}$. Med det samme $\mathscr{H}$, ved at bruge forskellige $\mathscr{A}$, kan vi producere forskellige $g^{\mathscr{D}}$, og siden $g^{\mathscr{D}}$ er byggestenen for bias-variance analyse, giver dette forskellige bias og var værdier
- Selvom bias og var analyse er baseret på squared error measure, behøve $\mathscr{A}$ ikke selv være baseret på minimeringen af denne. Den kan producere $g^{\mathscr{D}}$ som den ønsker, men når den er færdig vurdere vi bias og var udfra squared error.

Beklageligvis kan vi ikke beregne bias og variance i praksis, siden de afhængde af vores target funktions og inputtet sandsynligheds fordeling, som vi ingen af delene kender. Dermed er bias-variance kun et konceptuelt værktøj, som er brugbart når vi skal udvikle en model. Vi vil gerne opnå en lav variance og en lav bias. Disse kan opnås med heuristiske tiltag som regularisering.

LEARNING THEORY - REGULARIZATION AND VALIDATION

Regularization er en måde, at ændre på inlærings algoritmen, hvor ved man kan bekæmpe overfitting, ved at *straffe* inlærings algoritme for at øge kompleksiteten af den hypothese, og dermed *foretrække* mindre komplekse hypotheser.

Overfitting er hvis vores hypothese tilpasser sig vores test data så meget, så E_{in} er meget lav, men E_{out} vil stige. Dette skyldes, at vi begynder at lære *noice* i vores test data. En af grundene til, at vi vil overflødig kompleksitet til live, er at mængden af testdata krævet for, at få ordenlig lærings resultater, bliver størrer med mængden af dimensioner, som vi ved fra VC analyse.

En brugt metode indenfor regularization er ved brug af weight decay, hvor vi vil minimere $E_{in} + \omega(h)$, hvor vi vil opnå optimale resultater med så små weights som muligt. Vi definere $\omega(h(\mathbf{w})) = \lambda||\mathbf{w}||_2^2 : \lambda > 0$.

En anden brugt metode er constrained optimization, hvor vi forsøger at

$$\begin{aligned} &\text{minimize} && \frac{1}{|D} \sum_{x,y \in D} (\mathbf{w}^T \mathbf{x} - y)^2 \\ &\text{subject to} && \lambda||\mathbf{w}||_2^2 \leq C \end{aligned}$$

Denne form for constrain kaldes for soft order constraint, fordi den kun *opfordre* alle weights til at være små, uden at ændre på ordenen af polynomiet ved at sætte visse indgange i $\mathbf{w}$ til 0. Disse constraints sætte så ift en *budget* C, hvor et større C giver mindre regularization, og et mindre C giver mere regularization. Vi har yderliger med denne form for constraint at $\mathcal{H}_{Constraint} \subseteq \mathcal{H} \implies d_{VC}(\mathcal{H}_{Constrained}) \leq d_{VC}$.

Det skal understreges, at, som bogen siger det, regularization er et nødvendigt onde. Det er mere en kunst end videnskab, og er derfor meget præget af heuristik. Men hele humlen ift de to regularization metoder vi har snakket om, falder på valget af det korrekte λ, som validation, bl.a., kan hjælpe os med.

Hvis vi ser på hvad regularization og validation forsøger at beskrive kan det ses som

$$\underbrace{E_{out}(h)}_{\text{validation estimere dette}} = E_{in}(h) + \underbrace{\omega(h)}_{\text{regularization estimere dette}}$$

Til validation bruger vi *validation sets*, som kan tænkes på som udpluk af vores test data, som vi gemme fra inlærings algoritmen, for så senere, at se hvordan den præstere på dette data. Det er vigtigt at dataen tages fra, inden vi begynder algoritmen, da den eller ville have lært fra dataen, som vi bruger som et *uafhængigt* estimat af E_{out}.

Lad vores test data være $N - K$ og validation mængden være K. Vi definere

$$E_{val}(g^-) = \frac{1}{K} \sum_{\mathbf{x}_n \in \mathcal{D}_{\text{val}}} e(g^-(\mathbf{x}_n), y_n)$$

Og kan beviser med VC bound for en endelig model med en hypothese is sig, og se med højsandsynlighed at

$$\begin{aligned} &\text{Hvis classification} && E_{out}(g^-) \leq E_{val}(g^-) + O\left(\frac{1}{\sqrt{K}}\right) \\ &\text{hvis regressin} && E_{out}(g^-) \leq E_{val}(g^-) \pm O\left(\frac{\sigma}{\sqrt{K}}\right) \end{aligned}$$

Så validation kan hjælpe os, med at give et but på hvad E_out vil blive, hvilket er uhyggelig vigtigt, da dette giver os en måde at vurdere vores model, og alle de valg vi har truffet igennem processen med at lave den, ved at give os et sammenligningsgrundlag. Men denne egenskab stiller os bos i et dilemma, da jo bedre estimat vi skal have, jo mere data skal vi tage til siden, til validation mængder, som så ikke kan bliver brugt på læring. Dette er beskrevet i bogen som

$$E_{out}(g) \underset{\text{(lille K)}}{\approx} E_{out}(g^-) \underset{\text{(Stort K)}}{\approx} E_{val}(g^-)$$

En måde at afhjælpe dette dilemma på er igennem cross validation, vi tager et udtræk af dataen,fx 1 point, og beregner

$$E_{CV} = \frac{1}{N}\sum_{n=1}^{N} e_n$$

hvor $e_n = e(g_n^-(\mathbf{x}_n), y_n)$, altså fejlen lavet for det udtrukkede punkt x_n. Vi lader nu

$$\bar{E}_{out}(N) = \mathbb{E}_{\mathscr{D}}[E_{out}(g)]$$

være det forventede, over data sets $\mathscr{D}$ af størrelse N, af out-of-sample error lavet af modellen. Så har vi

THEOREM 4.4 *E_{CV} er et unbiased estimat af $\bar{E}_{out}(N-1)$. Den forventede model performance, $\mathbb{E}[E_{out}]$, over data sets af størrelse $N-1$.*

Så fx ift vores λ kan vi nu, laver en mængde λ'er og prøve dem af, og ved hjælp af cross validation, vælge de bedste λ, for så til sidst at træne modellem på hele data mængden.

SUPPORT VECTOR MACHINES - KERNELS

Support vector machines, er blandt de bedste, "of the shelf" supervised learning algorihtms, der findes. SVM's bruges til classification, og vi følgende notation, hvor $y \in \{0,1\}$ og bruge en w,b notation til, betegne vores hyperplane. Vores klassifier er så ledes

$$h_{w,b}(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x}+b)$$

Dermed er $g(z) = \Longrightarrow \begin{cases} 1 & z \geq 0 \\ -1 & z < 0 \end{cases}$ Til SVM benytter vi en margin begreb, både i en functionel og en geometrisk forstand. vi definere **functionel margin** for (w,b) til

$$\hat{\gamma}^{(i)} = y^{(i)}(\mathbf{w}^T\mathbf{x}+b)$$

Så jo mere sikkert vi er på $y=1$ højere bliver vores margin, og ligeså for $y=-1$. Funktionel margin har dog den lidt uheldige egenskab af fx $g(\mathbf{w}^T\mathbf{x}+b) = g(2\mathbf{w}^T\mathbf{x}+b)$, hvilket betyder, at $h_{(w,b)}$ kun afhænger af fortegnet og ikke størrelsen af $\mathbf{w}^T\mathbf{x}+b$. Intuitivt vil det derfor give mening af lave en normaliserings condition.

På samme tid vil vi også definere en **geometrisk margin**. Denne skal ses som den orthogonale længde fra vores hyperplane til hvert punkt. Det klart som med den functionelle margin, at jo større margin, jo mere sikker er vi på, at data punktet er klassificeret korrekt. Vi definere den geometriske margin sålede

$$\gamma^{(i)} = y^{(i)}\left(\left(\frac{w}{||w||}\right)^T x^{(i)} + \frac{b}{||w||}\right)$$

Vi ser at hvis $||w|| = 1$ er den geometriske margin, den samme som den functionelle.

For begge marginer definere vi, givet et træningssæt $S = \{(x^{(i)}, y^{(i)}); i = 1,...,m\}$

$$\hat{\gamma} = \min_{i=1,...,m} \hat{\gamma}^{(i)}$$

som værende den mindste margin i træningssettet.

Det vi reelt ønsker af vores SVM er at maximere længden fra de nærmeste punkter til vores hyperplane, så vi har den "mest sikre" klassificering af dataen, dette giver følger optimerings problem

$$\max_{\gamma,w,b} \quad \gamma \tag{0.1}$$

$$\text{s.t.} \quad y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)}+b) \geq \gamma \qquad i = 1,...,m \tag{0.2}$$

$$||w|| = 1 \tag{0.3}$$

men grundet $||w|| = 1$ ikke er en convex function, kan dette lade sige gøre, så vi omskrive problemet

$$\max_{\gamma,w,b} \quad \frac{\hat{\gamma}}{||w||} \tag{0.4}$$

$$\text{s.t.} \quad y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq \hat{\gamma} \qquad i = 1,...,m \tag{0.5}$$

$$\tag{0.6}$$

men nu har vi at $\frac{\hat{\gamma}}{||w||}$ ikke er convex, derfor omskriver vi igen så

$$\max_{\gamma,w,b} \quad \frac{1}{2}||\mathbf{w}||^2 \tag{0.7}$$

$$\text{s.t.} \quad y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 \qquad i = 1,...,m \tag{0.8}$$

$$\tag{0.9}$$

hvilket vi kan fundet den vilkårlige skalering af (w, b) ikker havde nogen indflydelse, hvorved vi kan sætte $\hat{\gamma} = 1$, og resultatet følger, derfra.

Vi kunne stoppe her, og løser problemet med quadratic programming, men der er flere ting der kan gøres bedre, fx hvad hvis dataen ikke er lineær seperable, eller vi ønsker en ikke linear opdeling af dataen?

Vi ser med ovenstående maximerings problem at vi kan skrive en konstraint for hvert training exempel

$$g_i(\mathbf{w}) = -y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) + 1 \leq 0$$

Fra KKT dual complementarity condition, vil vi så have at $\alpha_i > 0$, kun for de træning exempler som har en functions margin, som er præcis lig 0, altså $g_i(\mathbf{w}) = -y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) + 1 = 0$. Altså vil de exampler med en optimal løsning være de punkter der er tættest på vores hyper plan. Disse kaldes for **support vectore**.

Efter en længere udredning vil vi se at man i langrage dual problemet vil kunne funde α_i, at vi kan beregne

$$\mathbf{w}^T\mathbf{x} + b = \left(\sum_{i=1}^{m} \alpha_i y^{(i)}\mathbf{x}^{(i)}\right)^T \mathbf{x} + b = \sum_{i=1}^{m} \alpha_i y^{(i)}\langle \mathbf{x}^{(i)}, \mathbf{x}\rangle + b$$

Hvor største delen af α_i vil være 0 på nær dem som er vores support vectorer, og derfor vil mange af ledende være 0 i ligningen. Dette er det som er kendt som support vector machines.

Givet at vi ønsker at checke vores data for en nogle bestemte feautre givet ved $\phi(\mathbf{x}) \begin{bmatrix} \mathbf{x} \\ \mathbf{x^2} \\ \mathbf{x^3} \end{bmatrix}$ kan vi, da vores SVM algoritme er opbygget af indreprodukter $\langle \mathbf{x}, \mathbf{z}\rangle$, og udskifte disse med

$\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$. Mere specifikt, given en feature mapping ϕ, definere vi Kernel

$$K(x,z) = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

Denne kerne udskifter blot alle de indre produkter i vores algoritme. En interessant egenskab ved vores Kernel, er at selvom $\phi(\mathbf{x})$ kan være meget *dyr* at udregne kan, Kernel være meget *billig*. Dermed kan vi med en effektiv måde at udregne $K(\mathbf{x},\mathbf{z})$ på, lære high dimensional feature space, givet ϕ, uden explicit at udregne $\phi(\mathbf{x})$. Faktisk kan vi se at et high dimensional $\phi(\mathbf{x})$ kræver $O(n^2)$ tid, hvorimod, at finde $K(\mathbf{x},\mathbf{z})$ kun tager $O(n)$ tid, altså linear tid.

Men hvornår er en kernel en valid kernel? Lad os starte med at definere en Kernel matrix, som er en $m \times m$ matrix K for et træningsset $\{\mathbf{x}_1,..,\mathbf{x}_m\}$, med indgang i,j givet ved $K_{ij} = (\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. Hvis K var en valid kerne, så følger $K_{ij} = K(\mathbf{x}^{(i)}, \mathbf{z}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(j)})^T \phi(\mathbf{x}^{(i)}) = K(\mathbf{x}^{(j)}, \mathbf{z}^{(i)}) = K_{ji}$ hvilket betyder at K er symmetrisk. Yderlige har vi brug for at K er semi-definite, altså for en arbitrær vektor $\mathbf{z}$ gælder, $\mathbf{z}^T K \mathbf{z} \geq 0$.

THEOREM (MERCER) Lad $K : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ være givet. For at K er en valid (Mercer) kernel, er det nødvendigt, og tilstrækkeligt at for vilkårlig $\{\mathbf{x}^{(1)},...,\mathbf{x}^{(m)}\}, (m < \infty)$, er den tilsvarende kernel matrix symmetrisk og positiv semi-definite.

Ligesom andre lærings algoritmer, kan vi også bruge regularization med SVM,. Denne defineres som

$$\begin{aligned}
\min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{m} \xi_i & \\
\text{s.t.} \quad & y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1 - \xi_i & i = 1,...,m \\
& \xi_i \geq 0 & i = 1,...,m
\end{aligned}$$

Dermed kan vi have functionelle margins der er mindre end 1 nu, og hvis vi har en functionmal margin $1-\xi_i$ (for $\xi > 0$), ville vi betale med en stigning i objective funktionen svarende til $C\xi$. Dette betyder C styrer den relative vægtning af målet om at lavet $||w||^2$ lille og forsøge at opnå en functional margin på mindst 1.

Vores dual problem vil da blive

$$\begin{aligned}
\max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle & \\
\text{s.t.} \quad & 0 \leq \alpha_i \leq C & i = 1,...,m \\
& \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 &
\end{aligned}$$

NEURAL NETS - BACKPROPAGATION

Et neural network, er en samling af neuroner, som er en beregnings enhed som tager, fx input x_1, x_2, x_3 og et +1 intercept term, også kaldet en bias unit, og som har output $h_{W,b}(\mathbf{x}) = f(W^T\mathbf{x}) = f(\sum_{i=1}^{3} W_i x_i + b)$, hvor $f : \mathbb{R} \mapsto \mathbb{R}$ kaldes for activation function. Denne function $f(\cdot)$ kan defineres som ønsker, fx som sigmoid funktionen

$$f(\mathbf{z}) = \frac{1}{1+\exp(-\mathbf{z})}$$

Et neural network, er så bygget op i flere lag af sådanne neuroner. Disse er ofte opdelt i et L_1 input layer, $\{L_2, ..., L_{m-1}\}$ hidden layer og L_m output layer.

Vi benytter følger notation om activitions, output value $a_i^{(l)}$ for enhed i i layer l. Lad $z_i^{(l)}$ være den total vægtede sum af input til enhed i i layer j, fx $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}$, sådan at $a_i^{(l)} = f(z_i^{(l)})$.

Til forskellige network kan vi vælge forskellige **artikture**, som har varierende antal hidden layer, input units, og outputs units. Vi definere et **feedforward** neural network, som en connectet graph uden direkte loops eller cykler.

Lad os antage vi har et fixed training set $\{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(m)}, y^{(m)})\}$ af m trænings exempler. Så kan vi træne et neural network ved brug af batch gradient descent. Så for et givent trænings example (x, y), kan vi definere en cost function, nærmere bestemt en one-half squared-error cost function

$$J(W, b; w, y) = \frac{1}{2}||h_{W,b}(x) - y||^2$$

Given et træningsset af m exempler, kan vi definere the overall cost function som

$$\begin{aligned} J(W,b) &= \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{jl}^{(l)}\right)^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}|\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{jl}^{(l)}\right)^2 \end{aligned}$$

hvor den først summand er vores average sum-of-squares error, og den anden summand er regularization term, også kendt som *weight decay*, som hjælper imod overfitting. Sådan et netværk kan både bruges til classification og regressions problemer.

Vores mål er at minimere $J(W, b)$, som en function af W og b. Så for at træne netværket initialisere vi hvert parameter $W_{ij}^{(l)}$ og hvert b_i^l, med små tilfældige værdier, tæt på 0, over fx en normal fordeling. Denne tilfældighed skal være *symmetry breaking*. Her efter bruger vi en optimerings algoritmer, så som batch gradient descent. Siden $J(W, b)$ ikke er convex,

vil vi højst sandsynlig havne i et lokalt minimum, men dette virker i praksis, ganske fint. En iteration af gradient descent updatere W, b parametrene som følger

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$
$$b_i^{(l)} := b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

Backpropagation er en algoritme til, effektivt at beregne gradient descent for et neural net.Intuitionen bag algoritmen er som følger. Given et træning exemple (x, y), kører vi først et "forward pass" som beregner all activations igennem netværket, samt værdien for hypothese $h_{W,b}(\mathbf{x})$. For hver node i i layer l, beregner vi et "error term" $\delta_i^{(l)}$, som måler hvor " ansvarlig" noden var for errors i vores output. Algoritmen er som følger

1. Lav et feed forward pass, som beregner activation for layer $L_2, L_3, ...$, til output layer L_{n_l}
2. For hver output enhed i i layer n_l, out layer, set

$$\delta_i^{(n_l)}) \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(\mathbf{x}) \right\|^2 = -(y_i - a_l^{(n_l)}) \cdot f'(z_i^{n_l})$$

3. For $l = n_l - 1, n_l - 2, ..., 2$
   a) For hver node i i layer l, set

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_l+1} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Beregn det ønskede partielle differentiale, som er givet som

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}$$

NEURAL NETS - DEEP NETS

Deep nets og basicly det vi bruger til deep learning. Deep learning forsøger at modelere en høj form af abstraktion i dataen. Vi kan tænke på et simpelt deep net bestående af to set neuroner, et set der modtager inputtet og sender en modificeret version videre til det næste set. I deep nets kan der være mange af sådanne layers imellem input og output layered. Disse layers består ikke af neuroner, men det kan dog hjælpe at tænkte lidt på det, på den måde. Disser layers hjælpe deep net algoritmer til at lave mange beregnings lag bestående af både linear og ikke linear transformationer.

Et sådan deep net er fx convolution neural networks, CNN. Vi bruger convolution networks, siden de syntes at kunne løse problemer, fx linear classification learning algorithms har haft problemer, fx strecthing, scaling eller rotation af vores input, hvis det fx er et 2d billede. CNN er generelt gode til beregne på data, som har en grid-like-topology. Ordet convolution i CNN, antyder blot at i et af vores layers vil der blive brugt convolution i stedet for matrix multiplikation fx. Convulution som vi bruger den er defineret som

$$s(t) = \int x(a)\, w(t-a)\, da = (a * w)(t)$$

hvor $x(a)$ er en måling/input med alder a a og $w(t-a)$ er weight/kernel i tid $t-$ alder. Outputtet $s(t)$ kaldes for vores feature map. Hvis input og kernel har flere dimensioner, kaldes de for tensors. Dette hjælper os fx med at analysere billede hvor der er flere axer, hvor vi får

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n)$$

Hvor vi kan se at convolutions i flere dimensioner er kommutativ.

$$S(j,i) = (K * i)(j,i) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$

Denne egenskab kommer da vi har *flipped* kernel'en ift inputtet. Vi bruger normalt, den sidste formel, fordi der er mindre variation i mængden af valide værdier for m og n. Vi gør opmærksom på at, visse libraries implementere en ligende function kaldet *cross-correlation*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Convolution udnytter tre vigtige idéer.

1. Sparse interactions / sparse connectivity
    - Dettte opnås ved at gøre Kernel mindre end inputtet. Dette betyder for et input m og ouput n, ed kørsel $O(m \times n)$, begrænser vi antallet af connections ved hvert output til k og får $O(k \times n)$
2. Parameter sharing / tied weights
    - " ... the value of the weight applied to one input is tied to the value of a weight applied elsewhere.

- reducere storage requirements a modellen til k parameter

3. Equivariant representation
    - equivariant betyder basicly $f(g(x)) = g(f(x))$
    - hvis vi leder efter noget i et billede, og vi ville finde det, og derefter flytte, ville man fx også kunne flytte det først området det er i først, også finde det

Et typisk later i et CNN består af tre faser.

1. Lav convolutions i parallel for at producere en mængde af linear activations
2. Hver linear activation kommer igennem en ikkelinear activation function, som fx rectified linear activation function $f(x) = \max(0, x)$. Dette kaldes også for detector stage.
3. pooling function, som modificere outputtet for layered. Fx max pooling rapportere det maximale output for alle naboer indenfor et rektangle af en given størrelse, 4×4, fx.

Pooling hjælper os med at gøre repræsentationen approximativt *invariant* til små translations af inputtet. Dette kan hjælpe og med at sige, at en egenskab er tilstede, men ikke nødvendigvis hvor den her. Pooling enheden der pooler over flere features som bliver lært med forskellige parametre, akn blive invariant overfor transformation, fx rotation af et tag.

NEURAL NETS - AUTOENCODER

Sparse autoencoder, er en learning algorithm, til at automatisk at lære features for unlabeled data.

DECISION TREES AND ENSEMBLE METHODS - BAGGING

I kurset har vi beskæftiget os med to former for trees. Regressions trees, og classification trees.

Vores algoritme for, at opbygge regressions trees skal automatisk kunne afgøre splitting variables og split points, samt hvilken topologi vores tree skal have. Antag at vi en partition i M regioner, $R_1, ..., R_M$ og vi modellere svaret som en konstant c_m i hver region

$$f(x) = \sum_{m=1}^{M} c_m I(x \in R_m)$$

Hvis vi antager, at vores minimerings kritere er sum of squares, $\sum (y_i - f(x_i))^2$, så ser vi hurtigt at det bedste $\hat{c}_m$ blot er gennemsnittet for y_i i region R_m.

$$\hat{c}_m = \text{ave}(y_i | x_i \in R_m)$$

det bedste binære partitions ift minimering af sum of squares, kan generelt ikke beregnes. Så vi må bruge en grædig algoritme. Vi ser på en splitting varibale j og et split point s, og definere et par af half-planes

$$R_(j,s) = \{X | X_j \leq s\} \qquad \text{og} \qquad R_2(j,s) = \{X | X_j > s\}$$

Så søger vi splitting variablen j og split punkt s som løser

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_1 \in R_2(j,s)} (y_i - c_2)^2 \right]$$

For hvilken som helst valg af j og s kan den indre minimerinsg løses ved

$$\hat{c}_1 = \text{avg}(y_i | x_i \in R_1(j,s)) \qquad \text{og} \qquad \hat{c}_2 = \text{avg}(y_i | x_i \in R_2(j,s))$$

Dette kan gøres ved at skanne igennem alle inputs og afgøre det bedte (j,s) par, der er muligt. Når vi har fundet det bedste split, partitionere vi dataen i to region, og gentager splitting for begge regioner.

Men hvor store skal vores trees blive? Til dette bruger vi *cost-complexity pruning*. Lad $T \subset T_0$ være et vilkårligt træ, som kan fås ved at prune T_0, dvs at, komlapse et antal interne, ikke terminerende nodes. Lad

$$\begin{aligned} N_m &= \#\{x_i \in R_m\} \\ \hat{c}_m &= \frac{1}{N_m} \sum_{x_1 \in R_m} y_i \\ Q_m(T) &= \frac{1}{N_m} \sum_{x_i \in R_m} (y_i - \hat{c}_m)^2 \\ C_\alpha(T) &= \sum_{m=1}^{|T|} N_m Q_m(T) + \alpha |T| \end{aligned}$$

Hvor $C_\alpha(T)$ er vores cost complixity criterion og $\alpha \geq 0$ som skal afspejle tradeoff imellem tree size og den evne til at fitte data. Store α giver små træer, og små α giver store træer.

Ønskes classicfication tree, skal impurity measure $Q_m(T)$ blot udskiftes, med fx

Misclassification error	$\frac{1}{N_m}\sum_{i\in R_m} I(y_i \neq k(m)) = 1 - \hat{p}_{mk}$
Gini index	$\sum_{k\neq k'} \hat{p}_{mk}\hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1-\hat{p}_{mk})$
Cross-entropy eller deviance	$-\sum_{k=1}^{K} \hat{p}_{mk}\log\hat{p}_{mk}$

hvor $\hat{p}_{mk} = \frac{1}{N_m}\sum_{x_i\in R_m} I(y_i = k)$, for en klasse k observation i node m. Vi lader klassifikationen i en node være $k(m) = \arg\max_k \hat{p}_{mk}$

Bagging er også kendt som boostrap aggregation. Bootstrap er blot en metode, hvorved vi vælger flere set fra træningsdataen, og bruger den til at estimere forskellige ting. Fx estimering af prediction error

$$\widehat{Err}_{boot} = \frac{1}{B}\frac{1}{N}\sum_{b=1}^{B}\sum_{i=1}^{N} L(y_i, \hat{f}^{*b}(x_i))$$

Vi kan bruge en ligende ide til bootstrap for bagging average prediction over en samling a bootstrap samle, hvor med vi reducere variancen.

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x)$$

Overstående er et Monte Carlo estimat hvor det glæder for en fordeling $\widehat{\mathscr{P}}$, med sandsynlighed $\frac{1}{N}$ for data punkter (x_i, y_i), for $0 \leq i \leq N-1$, som har en "*true*" bagging estimat $E_{\widehat{\mathscr{P}}}\hat{f}^*(x)$. Vi ser at $\hat{f}_{bag}(x) \to \hat{f}(x)$ imens $B \to \infty$.

Dette kan også bruges i sammenhæng med en tree classifier $\hat{G}(x)$ for et K-class svar. Her vil vores bagged classifier vælge det svar med flest votes, og $\hat{G}_{bag}(x) = \arg\max_k \hat{f}_{bag}(x)$. Her er $\hat{f}_{bag}(x) = [p_1(x), p_2(x), ..., p_K(x)]$, hvor hver $p_i(x)$ er 0, ud over den med flest votes, der er 1.

Bagging fungere specielt godt med tree, som har høj varians, hvor bagging mindsker squared-error loss, og variance under at påvirker bias. Dog som vi sagde ovenfor, kan vi vise at, for $f_{ag}(x)$ som er bagging estimate

$$\begin{aligned} E_{\mathscr{P}}[Y - \hat{f}^*(x)]^2 &= E_{\mathscr{P}}[Y - f_{ag}(x) + f_{ag}(x) - \hat{f}^*(x)]^2 \\ &= E_{\mathscr{P}}[Y - f_{ag}(x)]^2 + E_{\mathscr{P}}[\hat{f}^*(x) - f_{ag}(x)]^2 \\ &\geq E_{\mathscr{P}}[Y - f_{agg}(x)]^2 \end{aligned}$$

Ovenstående holder dog ikke for classification under $0-1$, grundet nonadditivity for bias og variance. I det tilfælde vil det at bagge en dårlig classifier gøre den dårlige og en god classifier vil blive bedre.

Det skal dog bemærkes at når vi bagger en model, vil dens strukture gå tabt. Så et bagged tree er ikke længere et tree, fx.

HIDDEN MARKOV MODELS - BASIC ALGORITHMS AND APPLICATIONS

Markov Models bruges til analyse at sekventiel data. Vi atnager altså en fordeling af observationer som defineres som

$$p(\mathbf{x}_1,...,\mathbf{x}_N) = p(\mathbf{x}_1)\prod_{n=2}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1})$$

Dette betyder selfølgelig at den betingede sandsynlighed gives som

$$p(\mathbf{x}_n|\mathbf{x}_1,..,\mathbf{x}_{n-1}) = p(\mathbf{x}_n|\mathbf{x}_{n-1})$$

Hvis fordelingen for alle sådan betingede sandsynligheder er ens, kaldes modellen for en *homogeneous* Markov Model.

Det smart ved Markov Models er at, man kan antage at data'en er uafhængig på nær n-forgående observation. Fx vejret igår kan betyder noget for idag, men vejret fra et år siden, har måske ikke så meget med regning at gøre idag. Dette defineres for en 1st order Markov Model som

$$p(\mathbf{x}_1,...,\mathbf{x}_N) = p(\mathbf{x}_1)p(\mathbf{x}_2|\mathbf{x}_1)\prod_{n=3}^{N} p(\mathbf{x}_n|\mathbf{x}_{n-1},\mathbf{x}_{n-2})$$

Men dette har en pris. Antag at der er K parametre i for en observation, så vil $p(\mathbf{x}_n|\mathbf{x}_{n-1})$ være specificeret ved et set af $K-1$ parametre, for hver K states i $\mathbf{x}_{n-1}$ hvilket giver $K(K-1)$ parametre. Dette skalerer så for en Mth markov model til $K^M(K-1)$ paramtre, altså bliver det tungt at beregne, virkelig hurtigt.

Vi kan i stedet antage at der findes en underliggende model, hvor hvert datapunkt $\mathbf{x}_n$ afhænger en underliggende latent varibale $\mathbf{z}_n$ som kommer fra en underliggende state model. Dette medfører

$$p(\mathbf{x}_1,...,\mathbf{x}_n,\mathbf{z}_1,...,\mathbf{z}_n) = p(\mathbf{z}_1)\left[\prod_{n=2}^{N} p(\mathbf{z}_n|\mathbf{z}_{n-1})\right]\prod_{n=1}^{N} p(\mathbf{x}_n|\mathbf{z}_n)$$

Såfremt vores $\mathbf{z}_i$ er diskrete er den ovenstående model en hidden markov model HMM. Fordi hver $\mathbf{z}_i$ er ansvarlig for genereringen af hver deres $\mathbf{x}_i$ definere vi dem ud fra en 1-of-K lang binær vektor, definere vi matrix med fordelingerne $p(\mathbf{z}_i|\mathbf{z}_{i-1})$ som vores transition matrix, hvor $A_jk \equiv p(\mathbf{z}_{nk} = 1|\mathbf{z}_{n-1,j} = 1)$ er vores transitions sandsynligheder, hvor det gælder at $\sum_k A_{jk} = 1$, så A har $K(K-1)$ uafhængige parametre. Vi hår således at

$$P(\mathbf{z}_n|\mathbf{z}_{n-1},\mathbf{A}) = \prod_{k=1}^{K}\prod_{j=1}^{K} A_{jk}^{z_{n-1},jz_{nk}}$$

og den initial latent node $\mathbf{z}_1$ har $\pi_k \equiv p(z_{1k} = 1)$, sådan at

$$p(\mathbf{z}_1|\pi) = \prod_{k=1}^{K} \pi_k^{z_{1k}}$$

hvor $\sum_k \pi_k = 1$. Tilsidst definere vi ϕ som værende et set parametre, som beskrive fordelingen af $\mathbf{x}_i$, *emission probability*, hvor emission har formen

$$p(\mathbf{x}_n|\mathbf{z}_n,\phi) = \prod_{k=1}^{K} p(\mathbf{x}_n,\phi_k)^{z_{nk}}$$

Det hele samles og vi har da

$$p(\mathbf{X},\mathbf{Z}|\theta) = p(\mathbf{x}_n|\mathbf{z}_n,\phi) = p(\mathbf{z}_1|\pi)\left[\prod_{n=2}^{N} p(\mathbf{z}_n|\mathbf{z}_{n-1},\mathbf{A})\right]\prod_{m=1}^{N} p(\mathbf{x}_m|\mathbf{z}_m,\phi)$$

hvor $\mathbf{X} = \{\mathbf{x}_1,...,\mathbf{x}_N\}, \mathbf{Z} = \{\mathbf{z}_1,...,\mathbf{z}_N\}$ og $\theta = \{\pi,\mathbf{A},\phi\}$.

For at bruge en HMM skal vi bestemme *likehood* for en sekvens af observationer. Dette kan gøres ved hjælp af vertibi decoding. Lad $\mathbf{Z}^*$ være den mest sandsynlige *forklaring* på en række af $\mathbb{X}$, som er givet ved

$$\mathbf{Z}^* = \arg\max_{\mathbf{Z}} p(\mathbf{X},\mathbf{Z}|\theta)$$

Givet $\mathbf{X}$ og $\mathbf{Z}^*$, så kan vi

$$\begin{aligned} p(\mathbf{X},\mathbf{Z}^*) = \max_{\mathbf{Z}} p(\mathbf{X},\mathbf{Z}^*) &= \max_{\mathbf{z_1},...,\mathbf{z_N}} p(\mathbf{x}_1,...,\mathbf{x}_N,\mathbf{z}_1,...,\mathbf{z}_N) \\ &= \max_{\mathbf{z}_n} \max_{\mathbf{z}_1,...,\mathbf{z}_{N-1}} p(\mathbf{x}_1,...,\mathbf{x}_N,\mathbf{z}_1,...,\mathbf{z}_N) \\ &= \max_{\mathbf{z}_n} \omega(\mathbf{z}_N) \\ \mathbf{z}_N^* &= \arg\max_{\mathbf{z}_n} \omega(\mathbf{z}_N) \end{aligned}$$

Vertibi algoritmen kører rekursivt, hvor vi definere dens backtracking sålede,

$$\begin{aligned} &\text{base caes:} && \omega(\mathbf{z}_1) = p(\mathbf{x}_1,\mathbf{z}_1) = p(\mathbf{z}_1)p(\mathbf{x}_1|\mathbf{z}_1) \\ &\text{recursio:} && \omega(\mathbf{z}_n) = p(\mathbf{x}_n|\mathbf{z}_n)\max_{\mathbf{z}_{n-1}} \omega(\mathbf{z}_{n-1}p(\mathbf{z}_n|\mathbf{z}_{n-1})) \end{aligned}$$

og hvor algoritmen er defineret som

- $\omega[k][n] = 0$
- if $p(\mathbf{x}[n]|k) \neq 0$:
  - for $j = 1$ til K
    - if $p(k|j) \neq 0$
      - $\omega[k][n] = \max(\omega[k][n], \omega[j][n-1] \cdot p(x[n]|k) \cdot p(k|j))$

CLUSTERING AND OUTLIER DETECTION - REPRESENTATION BASED

Given et d-dimensionel rum, $\mathbf{D} = \{\mathbf{x}_i\}_{i=1}^n$, og given et antal ønskede clustere k, er målet med representative-based clustering at dele datasættet i k cluste $C = \{C_1, ..., C_k\}$. For hver cluster C_i findes der et repræsenterence punkt, kaldet en centroid μ_i, som er givet ved

$$\mu_i = \frac{1}{n_i} \sum_{x_j \in C_i} \mathbf{x}_j$$

Hvor $n_i = |C_i|$. En brute-force metode til dette problem er blot at checke alle clusters, som er givet ved *Stirling numbers of the secound kind*

$$S(n,k) = \frac{1}{k!} \sum_{t=0}^{k} (-1)^t \binom{k}{t} (k-t)^n$$

Hvilket giver at vi har $O(\frac{k^n}{k!})$ clusterings af n punkter i k grupper. I stedet for bruteforce, kan vi bruge K-means algoritmen. Her giver vi en score til hvert cluster ud fra square mean error

$$SSE(C) = \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in C_j} ||\mathbf{x}_j - \mu_i||^2$$

Hvor målet er at funde en clustering der minimere SSE

$$C^* = \arg\min_C \{SSE(C)\}$$

Algoritmen forløber så ledes, først initialiseres k tilfældige punkte som agere vores μ_i.Derefter tilskrives hvert punkt $\mathbf{x}_i$ til det cluster hvor afstand til dens centroid er minds. Når alle punkter er blevet assignet, udregner vi en ny centroid værdi for alle clusters, $\mu_i^t = \frac{1}{|C_i|} \sum_{\mathbf{x}_j \in C_i} \mathbf{x}_j$. Dette gentages indtil de nye centroid værdier for alle clusters er mindre end et valgt ϵ, altså $\sum_{i=1}^k ||\mu_i^t - \mu_i^{t-1}||^2 \leq \epsilon$. Fordelene ved K-means er at den er relativt hurtig $O(tkn)$, hvor t er antal iterationer, k er cluster og n er data punkter. Normalt er $t, k << n$, og den er nem at implementere. Men den har også en del ulemper. Den kan kun bruges hvis der er en måde at regne gennemsnittet ud på, vi skal på forhånd afgøre hvor mange klusters der er i datasettet, clusters fordeler alle punkter selvom det er støj eller outliers, og alle cluster bliver convexe.

K-mean er et eksempel på en *hard assignment clustering*, hvor hvert data point kun tilhører et cluster, lad os i stedet overveje en *soft assignment clustering*, hvor hvert data punkt tilskreves en sandsynlighed for at tilhøre et cluster. En sådan tilgang bruges i maximum likelihood estimation, MLE.

Givet et dataset $\mathbf{D}$, dfinere vi *likelihood* for θ, som en betinget sandysnlighed over $\mathbf{D}$ og θ

$$P(\mathbf{D}|\theta) = \prod_{j=1}^{n} f(\mathbf{x}_j)$$

Hvor $f(\mathbf{x})$ er givet ved en gaussian mixture model $f(\mathbf{x}|\mu_i, \Sigma_i)$. Vores mål med MLE er at vælge parametre θ, som maximere likelihood

$$\theta^* = \arg\max_{\theta}\{P(\mathbf{D}|\theta)\}$$

Da sandsynlighederne kan blive meget små bruges der normalt en *log-likelihood* function

$$\ln P(\mathbf{D}|\theta) = \sum_{j=1}^{n} \ln f(\mathbf{x}_j) = \sum_{j=1}^{n} \ln\left(\sum_{i=1}^{k} f(\mathbf{x}_j|\mu_j, \Sigma_i)P(C_i\right)$$

At maximere direkte på log-likelihood over θ er svært, istedet benytter vi *expectation-maximization*(EM). Algortimen er delt ind i tre faser,

1. **Initialization step**, hvor for hvert cluster C_i vælger vi tilfædigt et mean μ_i, hvor værdien μ_{ia} sættes for hver dimension X_a. Vi initialisere en covariance $d \times d$ matrix, $\Sigma_i = \mathbf{I}$, og sandsynligheden for at være i hvert cluster gives som $P(C_i) = \frac{1}{k}$.

2. **Expectation step**, hvor vi beregner sandsynligheden for et cluster C_i givet et punkt $\mathbf{x}_j$, ved brug af

   $$w_{ij} = P(C_i|\mathbf{x}_j) = \frac{f_i(\mathbf{x}_j) \cdot P(C_i)}{\sum_{a=1}^{k} f_a(\mathbf{x}_j) \cdot P(C_a)}$$

   som er væres wights.

3. **Maximization step**, given w_{ij} estimere vi igen Σ_i, μ_i og $P(C_i)$, givet ved

   $$\mu_i = \frac{\sum_{j=1}^{n} w : ij \cdot \mathbf{x}_j}{\sum_{j=1}^{n} w_{ij}} = \frac{\mathbf{D}^T \mathbf{w}_i}{\mathbf{w}_i^T \mathbf{1}}$$
   $$\Sigma_i = \frac{\sum_{j=1}^{n} w_{ij} \mathbf{z}_{ij} \mathbf{z}_{ij}^T}{\mathbf{w}_i^T \mathbf{1}}$$

   hvor $\mathbf{z}_{ij} = \mathbf{x}_j - \mu_i \in \mathbb{R}^d$, og

   $$P(C_i) = \frac{\sum_{j=1}^{n} w_{ij}}{n} = \frac{\mathbf{w}_i^T \mathbf{1}}{n}$$

De sidste to trin gentages intil $\sum_{i=1}^{k} ||\mu_i^t - \mu_i^{t-1}||^2 \leq \epsilon$.

REINFORCEMENT LEARNING - MARKOV DECISION PROCESS

I reinforcement learning, arbejde vi med to instancer. Agenten og Environmented. Agenten agere, udfører en action $A_t \in \mathscr{A}(S_t)$ i environmentet, som han så får feedback på, igennem en reward $R_{t+1} \in \mathscr{R} \subset \mathbb{R}$, som også kan være negativ. Dette spil kører igennem en tidsperiode, som kan være endelig, uendelig eller episodel. Vi beskriver denne tid i timestepd $t = 0, 1, 2, 3, ...$, som vi i dette kursus har holdt diskret, men der findes teori for continuel-time case. Vi definere vores mulige actions ud fra et state $S_t \in \mathscr{S}$

Målet med reinforcement learning er helt simplet, at *maximere vores expected value for summen af modtaget reward.* Vi definere

$$G_t \doteq R_{t+1} + R_{t+2} + ... + R_T$$

Hvor T er det sidste time step. Vi har dog et problem med denne formulering ift, hvis vi aldrig terminere, og her bruge *discountint,*

$$G_t \doteq R_{t+1} + R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

hvor $0 \leq \gamma \leq 1$ kaldes for discount rate. Hvis $\gamma = 0$ er vi kun interesseret i at optimere den umildbare reward, og jo tættere γ kommer på 1, jo mere tager vi fremtidige belønninger med i vores overvejelser. Hvis rækken af states er uendelig skal $\gamma < 1$ da vi ellers ville få $G_t \doteq \infty$.

Vi definere at et state signal har *Markov property*, hvis environmentets respons ved $t+1$ kun afhænger af state og actions repræsentationen ved t,

$$p(s', r|s, a) \doteq Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \qquad \forall r, s', s, a$$

En reinforcement learning task som opfylder Mrkov egenskaber siges at være en *Markov decision process*, MDP. Vi definere MDP som en fem tuppel $MDP = \{S, A, \{P_{s,a}\}, \gamma, R\}$, hvor

- S er mængden at states
- A er mængden af actions
- $\{P_{s,a}\}$ er state transitions probability som er givet ved $p(s'|s, a) \doteq Pr\{S_{t+1} = s'|S : t = s, A_t = a\} = \sum_{r \in R} p(s', r|s, a)$
- γ er discount factor
- og R er reward funktionen givet ved, $P_{s,a} : (S \times A) \to \mathbb{R}$

Til enhver reinforcement algoritme, skal vi give en strategi π, som mapper fra state til sandsynligheder over mulige actions. $\pi : S \to p(A)$, eller blot $\pi : S \to A$. Det vores strategi skal hjælpe os med, er at maximere vores reward function, over expected returns

$$R : S \to \mathbb{R} \qquad \mathbb{E}\left[\sum_{i=0} \gamma^i R(s_i)\right]$$

En måde vi bergener den på, er at følge vores strategi, og se hvad vi får ud af det. Dette definere vi som vores value function

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{i=0} \gamma^i R(s_i) | s_0 = s, \pi\right]$$

Udregningen af denne foregår rekursivt ved hjælp af Bellman ligningen

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\left[\sum_{i=0} \gamma^i R(s) | s_0 = s, \pi\right] \\ &= \mathbb{E}[R(s)] + \sum_{s' \in S} P_{s,\pi(s)}(s') \mathbb{E}\left[\sum_{i=1} \gamma^i R(s') | s_1 = s', \pi\right] \\ &= \mathbb{E}[R(s)] + \gamma \sum_{s' \in S} P_{s,\pi(s)}(s') V^{\pi}(s') \end{aligned}$$

Så det vi gerne vil finde er

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \mathbb{E}[R(s)] + \gamma \sum_{s' \in S} P_{s,a}(s') V^*(s')$$

Hvor vi grådigt vælger den optimale strategi π

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in S} P_{s,a}(s') V^*(s')$$

Derefter itererer vi over vores strategi på to forskellige måde, improvement i, og evaluation e. Vi stopper den iteration når vores strategi kun blive improvet mindre end Δ.