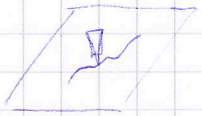
Mustererkennung

22.10.208

K-NN (K-nearest neighbour)

Auf einem Tablett schreibt man mit einem Stift und erhält dadurch Informationen von Position und Zeit (x, y, t)



Als Merkmale könnte man bei Zahlen z.B. die Koordinaten der Punkte als Vektoren darstellen.

Man könnte aber auch die Drehung bei der Stiftführung nehmen

$360°$ $0°$ $360°$

Jetzt kann man einen Vektor von Features machen, um alle Merkmale zur Erkennung zu kombinieren.

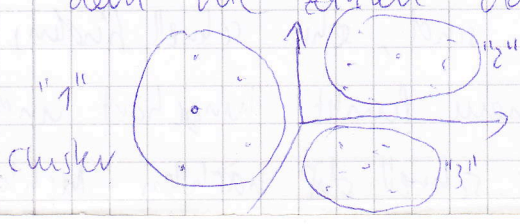
$$\vec{x} = \begin{pmatrix} \Theta \\ \alpha \\ L \\ \vdots \end{pmatrix}$$

Ein anderes Merkmal ist die Öffnung, der Abstand von Anfangs und Endpunkt geteilt durch die Länge des Zuges

$$\text{Öffnung} = \alpha = \frac{d}{L}$$

Um die Länge L als eigenes Merkmal einen Linienzuges zu nehmen, muss man allerdings die Länge normieren, z.B. auf Werte zwischen 0 und 1.

Bei d Features spannt sich ein Raum $\mathbb{R}^d$ auf, in dem die Zahlen dann durch Punkte identifizierbar sind, bzw. in bestimmten Bereichen landen.

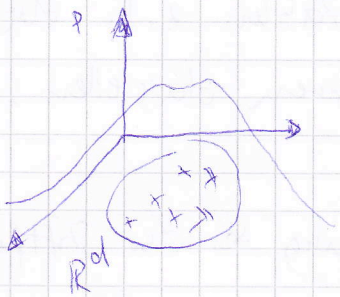


Trainingsmenge $N \gg 1$, die sich in 10 Klassen aufteilt.

Klassifizierung: [5] —Merkmalsextraktion→ $\vec{x} = \begin{pmatrix} \vdots \\ \vdots \end{pmatrix}$

Dadurch landet eine Zahl irgendwo im Raum, und dann stimmen die nächsten Nachbarn ab, zu welcher Klasse die Zahl gehört. Wenn man 3 Nachbarn fragt, nennt man es 3-NN.

1) Bayes-Ansatz. Man betrachtet nur eine Klasse hier.



Hier kann man jetzt über die Wahrscheinlichkeitsverteilung im Raum Aussagen über die Klassifizierung treffen.

Man könnte auch unstetige Verteilungen nehmen oder Gaußverteilungen. Das gute an Gauß ist, dass durch den Mittelpunkt und die Kovarianz die Verteilung eindeutig bestimmt ist.

Bei KNN sind die Daten selber die Wahrscheinlichkeitsverteilung.

2) K-NN ist teuer. Man muss für jedes neue $\vec{x}$ alle $d(\vec{x}, \vec{x_i})$ für alle N Testpunkte berechnen. Und unsere Trainingsmenge N ist sehr groß

Wie kann man K-NN jetzt besser bzw. schneller machen?

Random projections

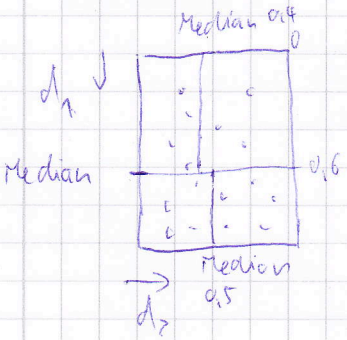
In unserem $\mathbb{R}^d$ erzeugen wir zufällig einen Vektor, der uns eine Richtung angibt. Dann werden alle Punkte auf den Richtungspfeil projiziert. Dann kann man wenn man jetzt die Punkte sortiert, die auf der Linie sind, sehr schnell finden, wo der neue Punkt hingehört und dann sehr schnell die nächsten Nachbarn

auf der Linie finden. Das macht man jetzt nochmal:

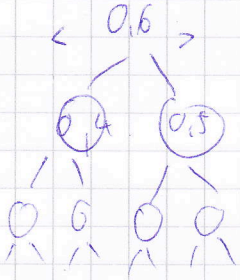
Die zweite Projektion kann man übrigens per Skalarprodukt machen

Ein anderes Verfahren, was besser zu machen wäre auch

Kd-Trees.



Man projiziert alle Punkte in jeweils nur eine Dimension und sucht den Median, d.h. bei welchem Punkt liegen genauso viele Punkte oberhalb und unterhalb.



Kd-Tree

Das macht man so oft, bis in jedem Kästchen nur 1 Punkt übrigbleibt.

Wenn jetzt ein neuer Punkt kommt, dann kann der Baum einfach durchlaufen werden, und man weiß genau, wo er drinn liegt. Dann weiß man immer nach $\sim \log_2 N$ in welcher Region man ist. Jetzt kann man wieder den nächsten Nachbarn mit Abstand bestimmen.

Eventuell muss man auch benachbarte Regionen betrachten.

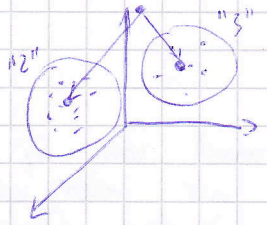
Dieses Verfahren ist aber auch nur gut, wenn die Dimension nicht so groß ist.

Musikerkennung
Vorlesung 3

29.10.2008

CLUSTERING

Man will, statt alle Datenpunkte zu behalten, für jede Klasse einen Vertreter haben.

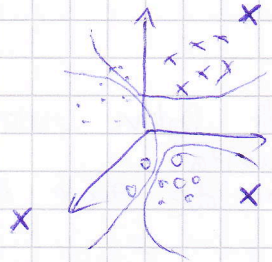


Ansonsten funktioniert es wie K-NN, bloß dass man bei Zahlen z.B. nur noch 10 Nachbarn betrachten muss.

Clustering ist nicht so mächtig wie K-NN, da man hier die Trennungslinien der einzelnen Cluster sehr einfach gestaltet. Dafür ist es schnell.

Unsupervised learning

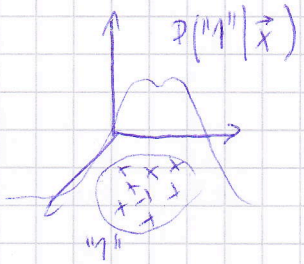
Wie können wir die Cluster bestimmen? Ein Algorithmus zum Clustering ist der LBG-Algorithmus. Wir gehen davon aus, dass wir k-Klassen im Datensatz haben.
A priori wählen wir k Vertreter $\vec{e}_1, \vec{e}_2, \dots, \vec{e}_k$ zufällig.



Dann berechnen wir die Einteilung der Daten auf die Vertreter, indem man pro Punkt guckt, welcher Vertreter am nächsten liegt. Das ist der sogenannte Expectation-Schritt.
Der Maximization-Schritt führt jetzt Korrekturen durch.

$\vec{e}_1 :=$ Schwerpunkt von Klasse 1
$\vdots$
$\vec{e}_k :=$ Schwerpunkt von Klasse k

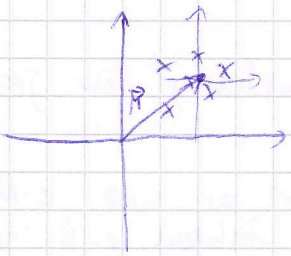
Jetzt iterieren wir über diese zwei Schritte, bis es keine Korrekturen mehr gibt, bzw. nach bestimmter Anzahl von Schritten. Der Algorithmus muss nämlich nicht terminieren.



Für K-NN wissen wir, dass die Daten selber ihre Wahrscheinlichkeitsverteilung bestimmen.
Für Cluster nehmen wir die Normalverteilung an, da wir ja nur einen Vertreter haben.

$$N(\vec{x}) = \|2\pi\Sigma\|^{-\frac{1}{2}} e^{-\frac{1}{2}(\vec{x}-\vec{M})^T \Sigma^{-1} (\vec{x}-\vec{M})}$$

Σ: Kovarianz



Als erstes zentrieren wir die Daten. Das funktioniert durch $\vec{x} - \vec{M}$. Danach ergibt sich sowas:

Wir wollen einen Abstand von $\vec{x}_p$ und $\vec{x}$ messen, dass machen wir aber nicht euklidisch, sondern mit dem Skalarprodukt.

$$S = \frac{1}{N}\left(\vec{x}^T \cdot \vec{x}_1\right)\left(\vec{x}_1^T \cdot \vec{x}\right) + \left(\vec{x}^T \cdot \vec{x}_2\right)\left(\vec{x}_2^T \cdot \vec{x}\right) + \dots$$

Die Formel sagt aus, wie gut der neue Vektor $\vec{x}$ zu den vorangehenden Daten passt.

$$S = \vec{x}^T \underbrace{\frac{1}{N}\left(\vec{x}_1 \vec{x}_1^T + \vec{x}_2 \vec{x}_2^T + \dots + \vec{x}_N \vec{x}_N^T\right)}_{\Sigma \text{ Kovarianzmatrix}} \vec{x}$$

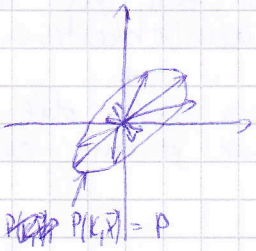
$$\Sigma = \frac{1}{N} \sum_{i=1}^{N} \left(\vec{x}_i \vec{x}_i^T\right)$$

In der Kovarianzmatrix, fassen wir alle Daten zusammen, und reduzieren dadurch die Komplexität.
Bei $N = 10^6$ und $d = 20$ hat die Matrix die Größe $20 \cdot 20 = 400$.

Für einen Cluster behalten wir also den Schwerpunkt $\vec{M}$ und die Kovarianzmatrix Σ.

Zurück zur Formel der Normalverteilung:

$$N(\vec{x}) = \underbrace{\|2\pi\Sigma\|^{-\frac{1}{2}}}_{\text{Normierung auf Volumen der Gaußglocke auf 1.}} e^{-\frac{1}{2}(\vec{x}-\vec{M})^T \Sigma^{-1} (\vec{x}-\vec{M})}$$



$$\vec{x}^T Q \vec{x} \geq 0$$

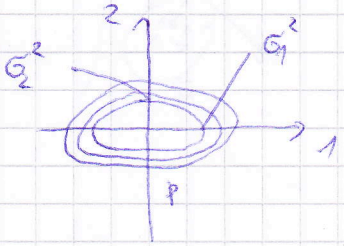
Was sagt uns der Teil $\frac{1}{2}(\vec{x}-\vec{M})^T \Sigma^{-1} (\vec{x}-\vec{M})$?
Dazu gucken wir uns mal einen Spezialfall an. Sei

$$\Sigma = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}$$

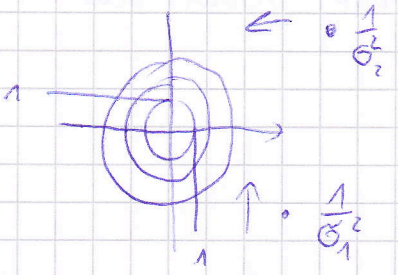
$$= \begin{pmatrix} \Sigma(x_{i,1} - \vec{M}_1)^2 & 0 \\ 0 & \Sigma(x_{i,2} - \vec{M}_2)^2 \end{pmatrix}$$

$$= \begin{pmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{pmatrix} \qquad \Sigma^{-1} = \begin{pmatrix} \frac{1}{\sigma_1^2} & 0 \\ 0 & \frac{1}{\sigma_2^2} \end{pmatrix}$$

Standardabweichung in Dimension 2



Durch Skalierung der Standardabweichung erhält man dann so ein Bild:



In einer Formel:

$$(\vec{x}-\vec{M})^T \begin{pmatrix} \frac{1}{\sigma_1^2} & 0 \\ 0 & \frac{1}{\sigma_2^2} \end{pmatrix} (\vec{x}-\vec{M})$$

Das ist genau so, wie in der Formel.

Wenn die Kovarianzmatrix nicht diagonal ist, dann kann man sie durch

$$\Sigma = R^T D R$$

diagonalisieren.?

Rotationsmatrix — Diagonalmatrix — Rotationsmatrix

$$\vec{x}^T \Sigma \vec{x} = (\vec{x}^T R^T) D (R\vec{x})$$

Dadurch löscht man die Korelation der Daten durch Dehnung der Achsen.

5.11.2008 Vorlesung 4

EM-Algorithmus

Wir machen es überwacht; d.h. wir betrachten die Tafel mit:

Wir berechnen die Stellvertreter der Klassen und berechnen dann pro Klasse die Kovarianzmatrix und daraus kann man dann die Wahrscheinlichkeitsverteilung bekommen.

Ein neuer Punkt wird so klassifiziert, dass man für jedes Cluster die Wahrscheinlichkeit an dem Punkt berechnet (durch die Formel von letzter Vorlesung) und dann die höchste nimmt und von dem Cluster das Label.

Spannender ist es, wenn wir unüberwacht arbeiten. Dann haben wir nur die Cluster ohne dessen Label zu kennen.

Wir machen jetzt folgendes: Wir wissen, dass es k Klassen sind und nehmen wie beim LBG zufällig k Stellvertreter $\vec{x}_1$ bis $\vec{x}_k$. Als Kovarianzmatrizen wählen wir die Identitäten.

1) Teilung der Daten per Wahrscheinlichkeit (Expectation)

$$f_i(\vec{x}) = \|2\pi\Sigma_i\|^{-\frac{1}{2}} e^{-\frac{1}{2}(\vec{x}-\vec{M})^T \Sigma_i^{-1} (\vec{x}-\vec{M})} \quad \text{mit } i = 1 \dots k$$

Die Klasse von $\vec{x}$ ist dann das i mit $\max(f_i, x)$

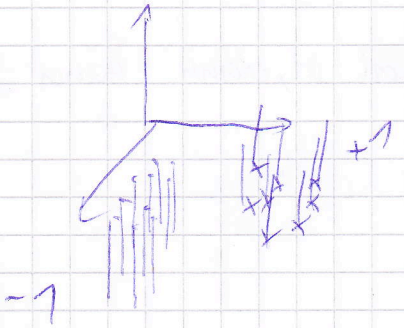
2) Berechne $\vec{M}_1, \vec{M}_2, \dots, \vec{M}_k$ neu anhand der Erzeugten Klassifizierungen. Die Kovarianzmatrizen müssen natürlich auch angepasst werden. (Maximization)

Das ganze iterieren wir nun wieder.

So ist das Clustering schicker, da auch die Form der Cluster mitberücksichtigt werden.

Jetzt was neues: <u>Lineare Regression</u>

Wir haben nun zwei Klassen



Wir hätten nun gerne eine Magicfunktion, die uns sagt, obein $\vec{y} = -1$ oder 1 hat.

Wir wollen, dass dieses f eine lineare Funktion ist. Das nennt man lineare Regression.

$$X = \begin{pmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1m} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2m} \\ \vdots & & & & \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{Nm} \end{pmatrix}$$

Datenmatrix

$$\vec{y} = X \cdot \vec{\beta} + \vec{\beta}_0$$

$\vec{y}$ ist die Klassifizierung, die für jeden Datenpunkt sagt, in welche Klasse der Punkt gehört.

$\vec{\beta}$ ist ein Vektor von Koeffizienten

Wie findet man nun $\vec{\beta}$? Wir setzen mal $\beta_0 = 0$.

$$\vec{y} = X \cdot \vec{\beta}$$

$$X^{-1} \vec{y} = X^{-1} X \cdot \vec{\beta} = \vec{\beta}$$

So leicht geht es leider nicht, da wir dazu eine quadratische Matrix bräuchten, dass muss es aber nicht geben.

Daher optimieren wir diesen Ansatz.
Wir bauen uns nun also eine quadratische Matrix

$$X^T \vec{y} = \underbrace{X^T X}_{\text{ist jetzt quadratisch}} \cdot \vec{\beta}$$

Das können wir nun invertieren; falls es existiert.

$$(X^T X)^{-1} X^T \vec{y} = (X^T X)^{-1} (X^T X) \vec{\beta} = \vec{\beta}$$

Falls unser $\beta_0 \neq 0$ ist, müssen wir den $\vec{y}$-Vektor zentrieren, dabei fällt dann β_0 weg und es geht weiter wie oben.

Falls wir also nicht $\vec{y} = X \cdot \vec{\beta}$ lösen können, suchen wir, ob wir eine Approximation minimal lösen können.

Dazu $Q = (\vec{y} - X\beta)^T (\vec{y} - X\vec{\beta})$ Quadratischen Fehler

$$\frac{dQ}{d\vec{\beta}} = -2X^T(\vec{y} - X\vec{\beta}) = 0$$

$$X\vec{y} = X^T X\vec{\beta}$$

$$\vec{\beta} = (X^T X)^{-1} X\vec{y}$$

ist dann wieder das selbe wie vorne.

Wie leitet man nun Vektoren ab?

$$\frac{d\,a}{d\vec{x}} = \begin{pmatrix} \frac{da}{dx_1} \\ \vdots \\ \frac{da}{dx_n} \end{pmatrix} \qquad \frac{d\vec{\beta}}{d\vec{\beta}} = \frac{d(\beta_1, \beta_2 \dots \beta_n)}{d\begin{pmatrix}\beta_1\\ \beta_2 \\ \vdots \\ \beta_n\end{pmatrix}} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & \vdots \\ \vdots & & \ddots & \\ 0 & 0 & \cdots & 1 \end{pmatrix} = I$$

wobei $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$

12.11.2008

Vorlesung 5

VORVERARBEITUNG DURCH PRINCIPAL COMPONENT ANALYSIS (PCA)

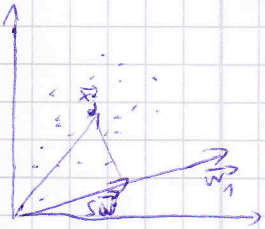
(Hauptkomponentenanalyse)

Wir wollen, dass nach der Vorverarbeitung die Daten nicht mehr korreliert sind. Man kann auch Datenkompression betreiben und es auch in der Computervision verwenden.

Wir haben hier also einen sichtbaren Cluster von Daten.
Die PCA soll uns nun gerade die Richtung aussprechen, in die das Cluster gerichtet ist

Dadurch ist es mir z.B. möglich, durch eine Hauptkomponente ein Cluster zu beschreiben.
Besser ist natürlich, wenn man so viele wie Dimensionen findet, geordnet nach Aussagekraft wie die Daten.

Algorithmus von Oja



- zufällig w, γ wählen (vorher müssen die Daten zentriert sein)
- $\vec{w}_1^{(i+1)} = \vec{w}_1^{(i)} + \gamma \langle \vec{w}_1^{(i)}, \vec{x}_j \rangle \left(\vec{x}_j - \langle \vec{w}_1^{(i)}, \vec{x}_j \rangle \vec{w}_1^{(i)} \right)$ $\quad 0 < \gamma \ll 1$

  $s := \langle \vec{x}, \vec{w}_1 \rangle$
- $\gamma^{(i+1)} = \gamma^{(i)} / 2$ oder anders verkleinern

Das Iterieren wir nun länger. Entweder, bis wir keine Zeit oder Lust haben oder bis $d(w)$ hinreichend klein ist.

Jetzt haben wir die 1. Hauptkomponente gefunden.
Um die nächste zu finden, löschen wir nun den Einfluss der Hauptkomponente aus den Daten.

$$\forall j \text{ do}: \quad \vec{x}_j^{(i+1)} = \vec{x}_j - \langle \vec{x}_j, \vec{w} \rangle \vec{w}$$

Usw. Für alle Hauptkomponenten.

Hauptkomponenten aus der Kovarianzmatrix

Mittelpunkt der Daten $\vec{\mu} = \frac{\sum_{i=1}^{N} \vec{x}_i}{N}$

$$\frac{\sum_{i=1}^{N} (\vec{w}^T \vec{x}_i - \vec{w}^T \vec{\mu})}{N} = \vec{w}^T \underbrace{\sum}_{\text{Kovarianzmatrix}} \vec{w}$$

Den Ausdruck wollen wir nun maximieren, mit der Nebenbedingung, dass $w^T w \stackrel{!}{=} 1$

$$z = \vec{w}^T \Sigma \vec{w} - \lambda (1 - \vec{w}^T \vec{w}) \qquad \text{(La Grange Multiplier)}$$

$$0 = \vec{w}^T \Sigma - \lambda \vec{w} \iff \Sigma \vec{w} = \lambda \vec{w} \qquad \text{Eigenwert}$$

Also maximiert $\vec{w}$ die Kovarianzmatrix, wenn $\vec{w}$ Eigenvektor von Σ ist.

$$w^T \Sigma w = \lambda$$

$\vec{w}$ muss Eigenvektor von Σ sein, damit die Varianz zu Maximieren.

λ_i zum i-ten Eigenvektor muss maximal gewählt werden.

Wenn wir jetzt die w_n n Hauptkomponenten haben, können wir die Korrelation entfernen.

$$\vec{x}^T \vec{w}_1 \; \vec{w}_1 = \vec{a}_1$$

$$\vec{x}^T \vec{w}_2 \; \vec{w}_2 = \vec{a}_2$$

$$\Sigma \vec{a}_i = \vec{x}$$

Wir drehen also das Koordinatensystem so, dass die Hauptkomponenten auf den Achsen liegen.

$$W = \begin{pmatrix} | & | & | & & | \\ w_1 & w_2 & w_3 & \dots & w_N \\ | & | & | & & | \end{pmatrix}$$

$$y = L W^T \vec{x} \qquad \text{mit} \qquad L = \begin{pmatrix} \frac{1}{\sqrt{x_1}} & & \\ & \ddots & \\ & & \frac{1}{\sqrt{x_n}} \end{pmatrix} \qquad \text{(Eye Sphering)}$$

Musterverkennung — FISHER - LINEAR - DISCRIMANT

Vorlesung 6

19.11.08

Bisher haben wir mehrere lineare Funktionen

$$f(x) = w^T x + w_0$$

implizit ~~zur~~ ~~klassifizieren~~ oder explizit betrachtet, um x zu klassifizieren.

Die Klassifizierungsregel ist

$$(1) \quad f(x) = \begin{cases} > 0 \Rightarrow x \in K_1 \\ < 0 \Rightarrow x \in K_2 \end{cases}$$

Beispiele

* k-nn

$$w = p_1 - p_2$$

$$w_0 = \frac{\|p_2\|^2 - \|p_1\|^2}{2}$$

p_i ist der Prototyp der Klasse K_i

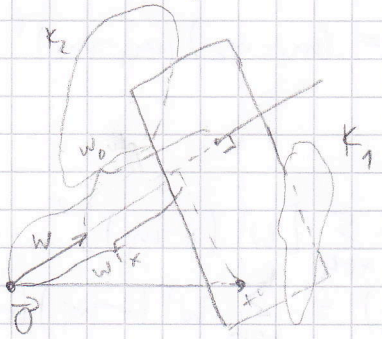
* Least Squares

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

$$w_0 = \frac{1}{N} \sum_{i=1}^{N} y_i$$

Annahme: $\|w\| = 1$

- Die Vektoren x, die $f(x) = 0$ erfüllen, definieren eine lineare Hyperebene in $\mathbb{R}^N$
- w ist orthogonal zur Hyperebene H
- w_0 ist der orthogonale Abstand zwischen Ursprung und H



Durch die Matrizenmultiplikation kann die Dimension auf eine Ebene reduziert werden.

Die Regel (1) sagt uns, auf welcher Seite der Ebene H der Vektor x liegt.

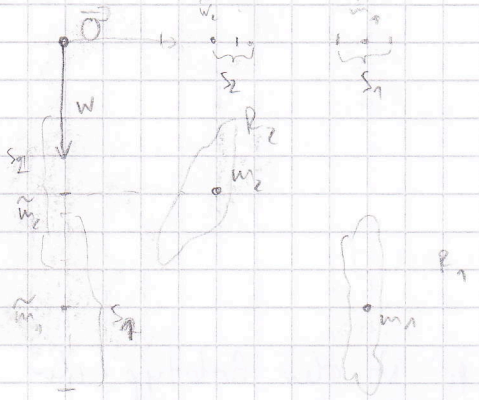
Dieser Ansatz versucht

- Probleme mit der Dimensionalität zu vermeiden, und zwar mit einer linearen Kombination (Transformation) der Merkmale + ein Qualitäts-Kriterium

Fisher: Finde die Richtung w, wo die Projektion der Daten „am besten" zu trennen sind!

Wie ist das Kriterium zu definieren?

Zuerst geometrisch...



$$\max_w (\tilde{m}_1 - \tilde{m}_2)^2 \quad \frac{S_i}{1} \text{ streuung}$$

Kriterium: Mittelwert und Varianz der Projektionen
groß $(\tilde{m}_1 - \tilde{m}_2)$
kleine Varianz

Jetzt algebraisch...

Mittelwert der Projektionen $\tilde{m}_i = \frac{1}{n_i} \sum_{x \in K_i} \tilde{x}_i$ mit $\tilde{x} = w^T x$ und

n_i Anzahl der Elemente in K_i

$$= \frac{1}{n_i} \sum_{x \in K_i} w^T x_i$$

$$= \frac{w^T}{n_i} \sum_{x \in K_i} x_i = w^T \left(\frac{1}{n_i} \sum_{x \in K_i} x \right) = w^T m_i$$

$$(\tilde{m}_1 - \tilde{m}_2)^2 = (w^T m_1 - w^T m_2)^2 = \left[w^T (m_1 - m_2) \right]^2$$

$$= w^T (m_1 - m_2)\, w^T (m_1 - m_2) = w^T (m_1 - m_2)(m_1 - m_2)^T w$$

$$= w^T S_B w \quad \text{mit} \quad S_B = (m_1 - m_2)(m_1 - m_2)^T$$

Varianz (oder etwas ähnliches) $s_i^2 = \sum_{x \in K_i} (\tilde{x} - \tilde{m}_i)^2$

$$= \sum_{x \in K_i} (w^T x - w^T m_i)^2 = \sum_{x \in K_i} \left[w^T (x - m_i) \right]^2$$

$$= \sum_{x \in K_i} w^T (x - m_i) w^T (x - m_i) = \sum_{x \in K_i} w^T (x - m_i)(x - m_i)^T w$$

$$= w^T \left(\sum_{x \in K_i} (x - m_i)(x - m_i)^T \right) w$$

$$= w^T S_i w \quad \text{mit} \quad S_i = \sum_{x \in K_i} (x_i - m_i)(x_i - m_i)^T \text{ der}$$

scatter Matrix. Das ist auch

$$S_i = n_i \Sigma$$

$(\tilde{m}_1 - \tilde{m}_2)^2 = w^T S_B w$ maximieren

$s_1^2 + s_2^2 = w^T S_1 w + w^T S_2 w = w^T S_W w$ mit $S_W = S_1 + S_2$ minimieren

Numerisches Kriterium:

(0) $\max_w J(w) = \dfrac{w^T S_B w}{w^T S_W w}$

Beobachtung: Gegeben $\alpha \neq 0$, es gilt ein

(*) $J(\alpha w) = J(w)$

Aus (*), das Optimierungsproblem (0) ist äquivalent zu

$\max w^T S_B w$ unter $w^T S_W w = 1$

Lagrange-Multiplikation $\Rightarrow \max_{w,\lambda} L(w,\lambda) = w^T S_B w - \lambda (w^T S_W w - 1)$

Finden wir w: $\dfrac{\partial L(w,\lambda)}{\partial w} = 2 S_B w - 2\lambda S_W w = 0$

$\Rightarrow S_B w = \lambda S_W w$

Das ist ein allgemeineres Eigenwertproblem.

$S_B w = \lambda S_W w$

$w = \frac{1}{\lambda} S_W^{-1} S_B w$ (skalar)

$= \frac{1}{\lambda} S_W^{-1} (m_1 - m_2)\underbrace{(m_1 - m_2)^T w}$

$= \frac{1}{\lambda} S_W^{-1} (m_1 - m_2)\, w^T (m_1 - m_2)$

$= \frac{1}{\lambda} S_W^{-1} (m_1 - m_2)(\tilde{m}_1 - \tilde{m}_2)$

$= \frac{\tilde{m}_1 - \tilde{m}_2}{\lambda} S_W^{-1} (m_1 - m_2)$

$J(w)$ ist maximal, wenn $w = \dfrac{(\tilde{m}_1 - \tilde{m}_2)}{\lambda} S_W^{-1} (m_1 - m_2)$ ist

$J(w) = J\left(\frac{\tilde{m}_1 - \tilde{m}_2}{\lambda} S_W^{-1} (m_1 - m_2)\right)$

$\overset{(*)}{=} J\left(S_W^{-1} (m_1 - m_2)\right)$

das heißt $w = S_W^{-1} (m_1 - m_2)$

Das wird behandeln wir im Tutorium...

$J(w)$ ist maximal, wenn

$$w = \frac{(\tilde{m}_1 - \tilde{m}_2)}{\nearrow} S_w^{-1} (m_1 - m_2)$$

ist.

$$J(w) = J\left(\frac{(\tilde{m}_1 - \tilde{m}_2)}{\nearrow} S_w^{-1} (m_1 - m_2)\right) \quad (*)$$

$$= J\left(S_w^{-1} (m_1 - m_2)\right)$$

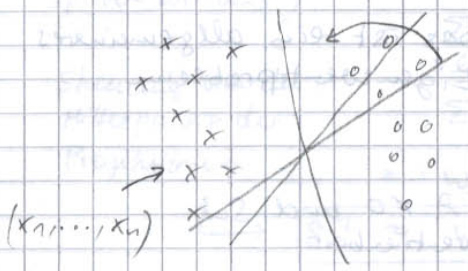
d.h. $\boxed{w = S_w^{-1} (m_1 - m_2)}$

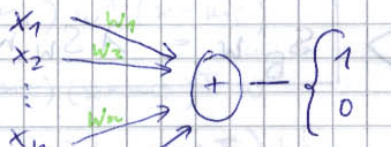
$f(x) = w^T x + w_0$ ← fehlt noch

26.11.08

Hatten schon: Lineare Trennung
- Fisher Diskriminante

Ähnliches Konzept: Perzeptron
- iteratives Verfahren
- anfänglich ausgewählte Trennung wird immer weiter verbessert





Lineare Kombination:

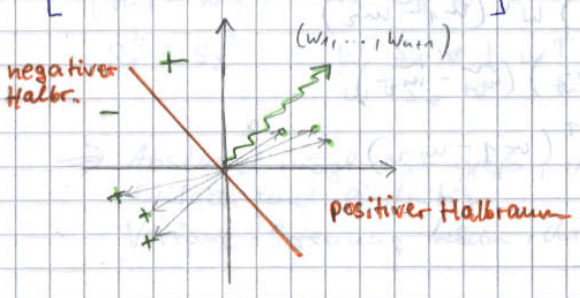
$w_1 x_1 + w_2 x_2 + \dots + w_n x_n + w_{n+1} \geq 0$

$\Rightarrow$ Ausgabe 1, sonst 0

↳ äquivalente Schreibweise:

$(x_1, x_2, \dots, x_n, 1) \cdot (w_1, \dots, w_{n+1}) \geq 0$

[$w_i x_i$ ist Multiplikation von Vektoren]

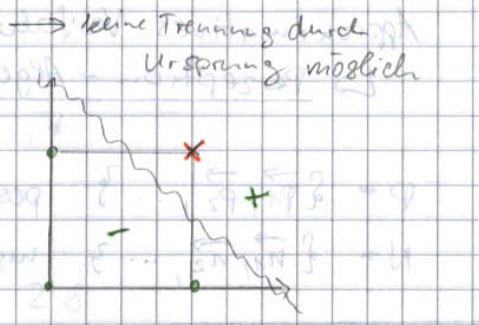


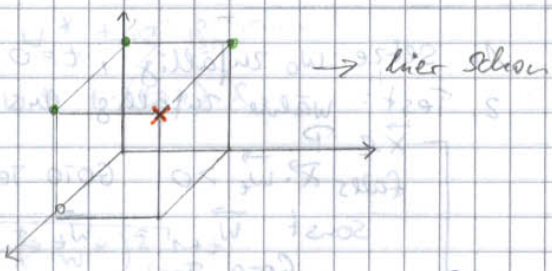
x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

Klassifizierung ↓

x_1	x_2	x_3	
0	0	1	0
0	1	1	0
1	0	1	0
1	1	1	1

zusätzl. Dimension einführen



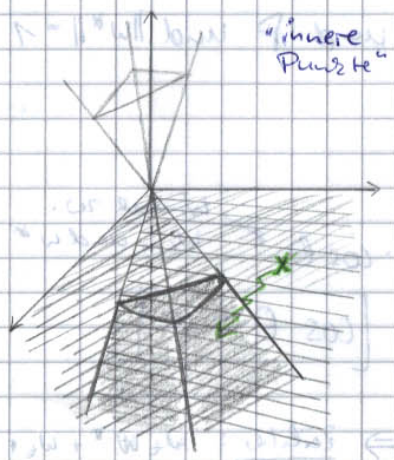


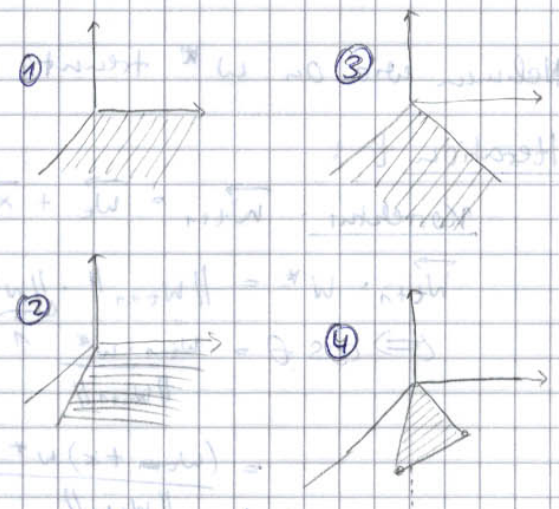
$$\underbrace{(w_1, w_2, w_3)}_{\text{unbekannt}} \cdot (x_1, x_2, x_3) \overset{?}{\geq} 0$$

	x_1	x_2	x_3	
①	$w_1 \cdot 0$	$+ w_2 \cdot 0$	$+ w_3$	< 0
②	$w_1 \cdot 0$	$+ w_2 \cdot 1$	$+ w_3 \cdot 1$	< 0
③	$w_1 \cdot 1$	$+ w_2 \cdot 0$	$+ w_3 \cdot 1$	< 0
④	$w_1 \cdot 1$	$+ w_2 \cdot 1$	$+ w_3 \cdot 1$	≥ 0

- w_3 muss negativ sein
- $w_2 + w_3$ muss negativ sein
- $w_1 + w_3$ muss negativ sein
- alle zusammen positiv

Lösung geometrisch:





- Lineare Programmierung zur Lösung bei viele Variable (schwierig)

oder

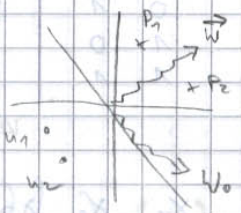
- Approximatives Verfahren

Approximatives Verfahren

↳ Perzeptron - Algorithmus

$P = \{\vec{p_1}, \vec{p_2}, \dots\}$ positive Vektoren

$N = \{\vec{n_1}, \vec{n_2}, \dots\}$ negative Vektoren



1. Setze w_0 zufällig, $t = 0$
2. Test: wähle zufällig aus $N \cup P$ ein $\vec{x}$

   $\vec{x} \in P$

   falls $\vec{x} \cdot \vec{w_t} > 0$ GOTO Test

   sonst $\vec{w}_{t+1} = \vec{w_t} + \vec{x}$, $t = t+1$

   GOTO Test

   $\vec{x} \in N$

   falls $\vec{x} \cdot \vec{w_t} < 0$ GOTO Test

   sonst $\vec{w}_{t+1} = \vec{w_t} - \vec{x}$, $t = t+1$

   GOTO Test

• positives Skalarprodukt heißt, $\vec{x}$ liegt auf der richtigen Seite der Trennungsebene ⇒ Winkel < 90

Wenn es Lösung gibt, konvergiert der Algorithmus, dauert aber womöglich sehr lange.

Nehmen wir an w^* trennt N und P und $\|w^*\| = 1$

Iteration t:

Korrektur: $\vec{w}_{t+1} = \vec{w_t} + \vec{x}$

$\vec{w}_{t+1} \cdot w^* = \|w_{t+1}\| \cdot \|w^*\| \cdot \cos\theta$ ← Winkel zw. w_{t+1} und w^*

$\Leftrightarrow \cos\theta = \frac{w_{t+1} \cdot w^*}{\|w_{t+1}\|}$ $\quad [\cos\theta \leq 1]$

$= \frac{(w_t + x) w^*}{\|w_{t+1}\|}$ ⇒ Zähler: $w_t \cdot w^* + w^* \cdot x$

$\geq w_t \cdot w^* + \delta$

$\delta = \min\{p_1 \cdot w^*, \dots, p_n \cdot w^*\}$

$\delta > 0$

Zähler:

$$w_t \cdot w^* = (w_{t+1} \cdot \vec{p}_i)\, w^*$$
$$= w_{t+1} \cdot w^* + \vec{p}_i \cdot w^*$$
$$> w_{t+1} \cdot w^* + \delta$$

$$\Rightarrow w_{t+1} \cdot w^* > w_{t+1} \cdot w^* + 2\delta$$
$$\geq w_{t+1} \cdot w^* + 3\delta$$
$$\vdots$$
$$> w_{t+1} \cdot w^* + (t+1)\,\delta$$

Nenner:

$$\|w_{t+1}\| = \sqrt{(w_t \cdot w^* + \vec{x} \cdot w^*)^2}$$
$$\vdots$$
$$\sqrt{\ldots (t+1) \ldots}$$

$\Rightarrow$ t kann nicht ∞ werden, irgendwo stoppt der Algorithmus

Mustererkennung

Vorlesung 8

3.12.-2008

Heute: Verallgemeinerung des Perceptron-Algorithmus

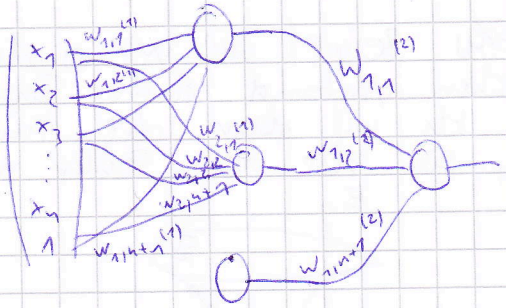
Hier wollen wir mehrere lineare Trennungen finden, die sich ergänzen. z.B. sinnvoll sei sowas:

$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ wollen wir klassifizieren.

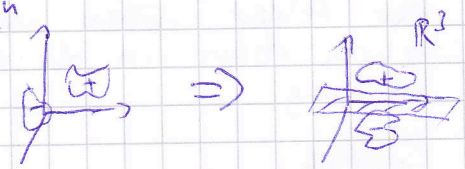
$\vec{x}$

Die $w_{ij}^{(k)}$ sind Gewichte.

Das nennt man dann „neuronales Netz"

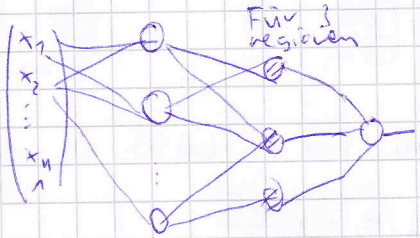


Was wir hier machen ist eine Projektion von $\mathbb{R}^n$ zu $\mathbb{R}^3$



Bei sowas

macht man das so:

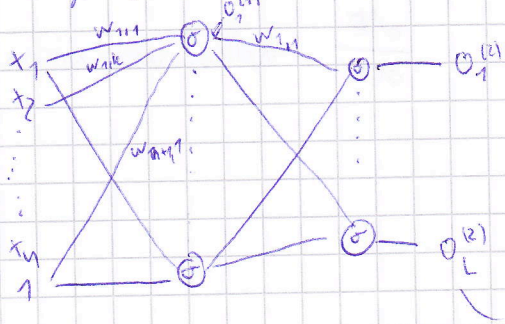


Wir machen das ganze stetig, so dass die Ausgabe einer Klassifikation nur 1 oder −1 ist, statt der normalen Signum-Funktion benutzen wir also

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

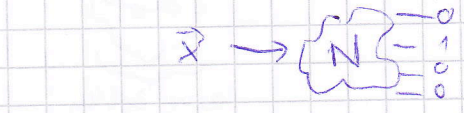
So sind die Ausgaben der Elemente "grauwertbehaftet", so dass man sie ein wenig mit Wahrscheinlichkeiten vergleichen kann.

Im Allgemeinen also:

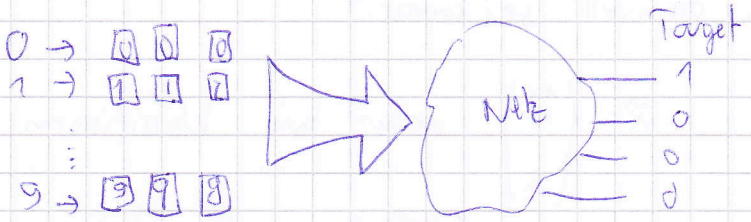


k Elemente L Elemente Mehrere Klassen.

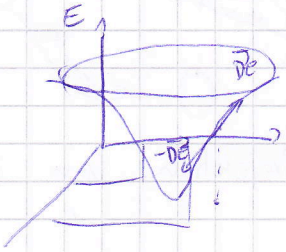
Bei Ziffern könnte es so aussehen:



Die Aufgabe ist jetzt, die Trainingsmenge so durch das Netz zu jagen, dass das Target erreicht wird.



Für jedes Beispiel, was wir so durchjagen, berechnen wir den euklidischen Abstand zum Target, der uns den Fehler verrät. Ich will dann die Kombination aus Gewichtsvektoren finden, für die der Fehler minimal ist.



Wir fangen mit zufälligen Gewichten an und schauen uns den Fehler an und gucken dann, in welche Richtung der Fehler höher wird, und gehen dann in die andere Richtung.

Fehlerfunktion ist Funktion aller Gewichte.

$$E(w_1, w_2, \ldots, w_m)$$

$$\vec{\nabla} E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_m}\right)$$

Die Korrektur für Gewicht w_i sei ΔE ist

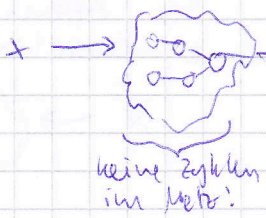
$$\Delta w_i := -\gamma \frac{\partial E}{\partial w_i}$$

Die Frage ist, wie man die partiellen Ableitungen bekommt.

wir verändern ein wenig w_1 aus $w_1 + \varepsilon$ und gucken was dann am, wie sich der Fehler ändert. $E \to E + e'$

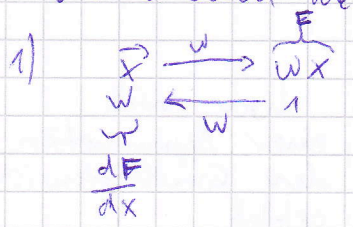
Dann gilt $\frac{\partial E}{\partial w_1} \approx \frac{(E+e') - E}{(w_1+\varepsilon) - w} = \frac{e'}{\varepsilon}$

Hilfsproblem:

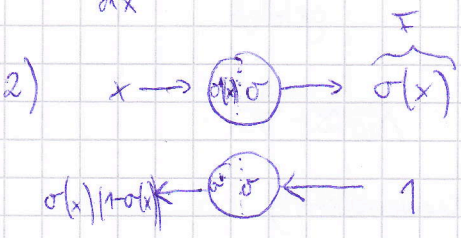


$\frac{\partial E}{\partial x}$ wollen wir wissen.

Wir unterscheiden mehrere Fälle: Wir rechnen mit Backprop

1) 

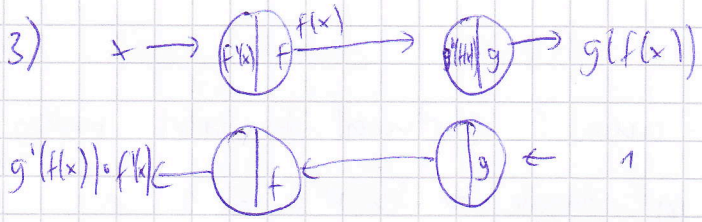
Feed forward
backprop

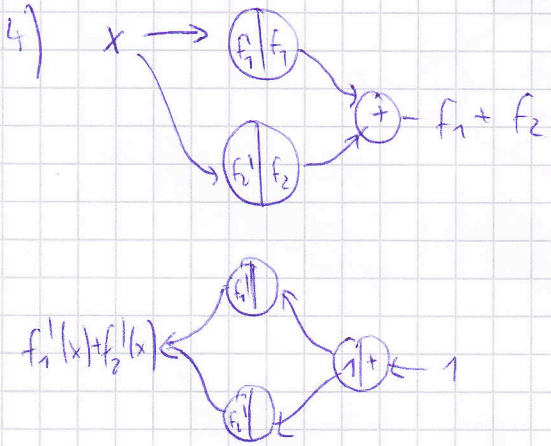
2) 

Feed forward

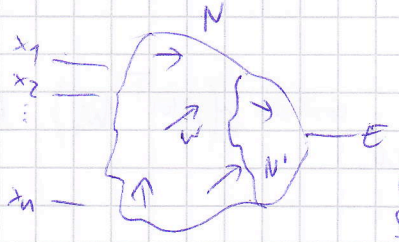
backprop. Die Regel ist hier: Linke Seite des Nehes verwenden und mit der Eingabe multiplizieren.

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

3) 

4) 

Jetzt können wir mit Backprop das Hilfsproblem alle lösen. Wieder zurück zum Hauptproblem.



Zunächst machen wir feed forward. Dann schreiben wir es wieder rückwärts, indem wir eine 1 reingeben.

Da wir keine Kreise haben, gibt es ein Subnetz N', das nur von der Veränderung beeinflusst wird. Man macht also beim Backprop nur N' betrachten und man ist fertig.

Das macht man für alle Gewichte dann. Und für alle Beispiele.

$$\frac{\partial E}{\partial w} = \frac{\partial (E_1 + E_2 \dots)}{\partial w}$$

Zwei Varianten von Backprop

Vorlesung 9

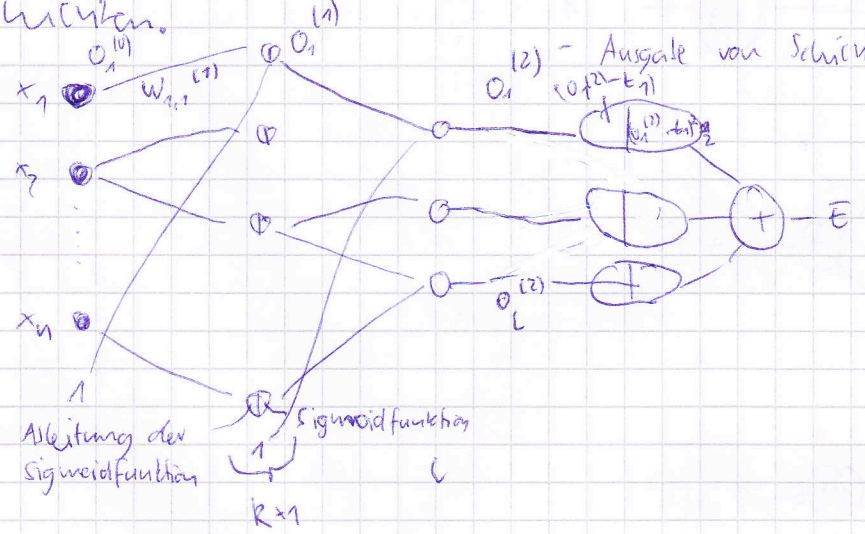
10.12.2008

1. On-line $(\vec{x}_1, \vec{t}_1)$ … $(\vec{x}_N, \vec{t}_N)$ $\rightarrow (\vec{x}_i, \vec{t}_i)$ $\Delta \vec{w} \approx \vec{\nabla} E_i$ hier wird einer Approximation der Gradientenrichtung gefolgt

2. Off-line (batch) $(\vec{x}_1, \vec{t}_1) \rightarrow E_1$, $(\vec{x}_2, \vec{t}_2) \rightarrow E_2$, … $(\vec{x}_N, \vec{t}_N) \rightarrow E_n$ E_n $\Delta \vec{w} \approx -\vec{\nabla} E$ Gradientenabstieg

Viele benutzen on-line Verfahren, da man hier nicht so schnell in ein lokales Minimum fällt, da man nicht direkt der Gradientenrichtung folgt.

Wie implementiert man das nun? Erstmal benutzt man Schichten.



W_1 W_2

Matrix der Gewichte mit $(n+1) \times k$

Der Feedforward Schritt besteht darin, dass man $s(\vec{O}^{(0)T} W_1)$ berechnet. Das gist die Ausgänge für die Hidden layer.

$$O_2^{(1)T} = s(\vec{O}^{(0)T} W_1)$$

Wenn man noch die Konstante 1 hinzufügt zum weiterrechnen nennt man das jetzt $\hat{O}^{(1)T}$. Für die letzte Schicht ist es dann

$$\vec{O}^{(2)T} = s(\hat{O}^{(1)T} W_2)$$

Wir müssen uns natürlich noch die Ableitungen mitspeichern. Das machen wir auch in Matrizen. D_1 für die 1. Schicht, D_2 für die 2.

$$D_1 = \begin{pmatrix} o_1^{(1)}(1-o_1^{(1)}) & & & 0 \\ & o_2^{(1)}(1-o_2^{(1)}) & & \\ & & \ddots & \\ 0 & & & o_k^{(1)}(1-o_k^{(1)}) \end{pmatrix}$$

$$D_2 = \begin{pmatrix} o_1^{(2)}(1-o_1^{(2)}) & & & 0 \\ & & \ddots & \\ & & & \\ 0 & & & o_l^{(2)}(1-o_l^{(2)}) \end{pmatrix}$$

Für den Fehler bzw. die Ableitung benutzen wir einen Vektor

$$\vec{\varepsilon} = \begin{pmatrix} o_1^{(2)} - t_1 \\ o_2^{(2)} - t_2 \\ \vdots \\ o_l^{(2)} - t_l \end{pmatrix}$$

$\vec{\delta}^{(2)} = D_2 \vec{\varepsilon}$. Jetzt können wir schon Korrekturen berechnen.

$\Delta w_{11}^{(2)} = -\gamma \delta_1^{(2)} o_1^{(1)}$

Jetzt für alle Gewichte.

$\Delta W_2^T = -\gamma \vec{\delta}^{(2)} \vec{o}^{(1)T}$, $\Delta W_2 = -\gamma \vec{o}^{(1)} \vec{\delta}^{(2)T}$

$\vec{\delta}^{(1)} = D_1 \hat{W}_2 \vec{\delta}^{(2)}$

W_2 ohne letzte Zeile !

$\Delta W_1^T = -\gamma \vec{\delta}^{(1)} \vec{o}^{(0)T}$

Man kann also mit nur ein paar Zeilen das Programmieren.

Jetzt kümmer wir uns um das γ. Für jedes w_{ij} haben wir ein γ_{ij}.

$$\gamma_i^{(k+1)} = \begin{cases} \gamma_i^{(k)} u & : \nabla_i E^{(k)} \cdot \nabla_i E^{(k+1)} \geq 0 \\ \gamma_i^{(k)} d & : \quad " \quad < 0 \end{cases}$$

$\Delta w_i^{(k)} = \gamma_i^{(k)} \nabla_i E^{(k)}$, $u = 1{,}5$ $d = 0{,}8$ oder so

(Algorithmus von Silva & Almeida)

Das heißt, wenn die partielle Ableitung $\frac{\partial E^{(k)}}{\partial w_i}$ in der k-ten und k+1-ten Iteration das gleiche Vorzeichen haben, dann wird γ größer gemacht.

Eine andere Heuristik wäre "Momentum".

$$\Delta w_i^{(k)} = -\gamma \frac{\partial E}{\partial w_i} + \alpha \Delta w_i^{(k-1)} \qquad \alpha < 1$$

RPROP - Algorithmus

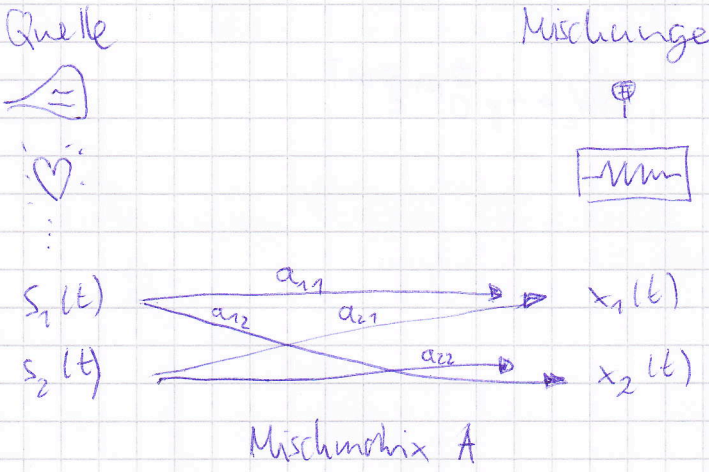
Es gibt eine Verbesserung für Silva & Almeida. Da ist das Problem, dass die Schritte die man macht, sehr klein um das Minimum werden. Zudem kann es passieren, dass man immer das Minimum überspringt.

$$\gamma_i^{(k+1)} = \begin{cases} \min(\gamma_i^{(k)} u, \gamma_{max}) : & \nabla_i E_i^{(k)} \nabla_i E_i^{(k+1)} > 0 \\ \max(\gamma_i^{(k)} d, \gamma_{min}) : & " \quad < 0 \\ \gamma_i^{(k)} & " \quad = 0 \end{cases}$$

$$\Delta^{(k)} w_i = -\gamma_i^{(k)} \operatorname{sgn}\left(\nabla_i E^{(k)}\right)$$

Blind source separation (ICA)

Vorlesung 7
17.12.2008



Mischmatrix A

Die Mischungen $x_i(t)$ sind eine lineare Funktion von dem Quellsignal $s_i(t)$

$$x_1(t) = a_{11} s_1(t) + a_{21} s_2(t)$$
$$x_2(t) = a_{12} s_1(t) + a_{22} s_2(t) \qquad (*)$$

$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ Mischmatrix. $\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} s_1(t) \\ s_2(t) \end{pmatrix}$

$$x(t) = A^T s(t)$$

Was ist jetzt das Blind-source-separation Problem?

Gegeben $x(t)$ mit $t = 1, \dots, T$. Finden den Quellsignale und die Mischmatrix.

Wir gucken es uns einmal geometrisch an:

Für jeden Zeitpunkt $t = 1, \dots, T$ betrachten wir den Vektor $x(t)$ als ein Datum, d.h. wir haben eine Trainingsmenge

$$\{x(1), x(2), \dots, x(T)\}$$

Nehmen wir an s_i ist dünnbesetzt, d.h. $s_i(t) \neq 0$ für wenige t.

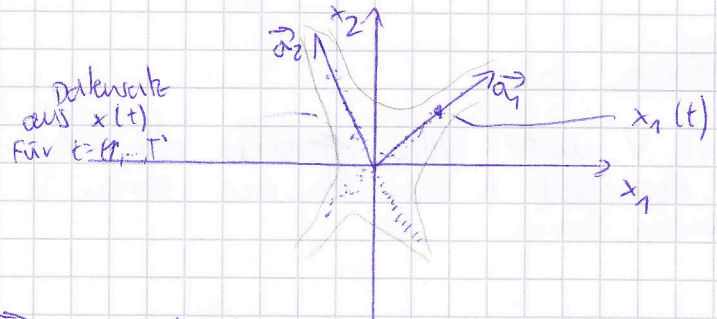
Das bedeutet,

$$s_1(t) \neq 0 \Rightarrow s_2(t) = 0 \quad \text{mit hoher Wahrscheinlichkeit}$$

Aus $(*)$ folgt dann

$$x_1(t) = a_{11}\, s_1(t) + 0 \qquad \vec{a}_1 = \begin{pmatrix} a_{11} \\ a_{12} \end{pmatrix}$$
$$x_2(t) = a_{12}\, s_1(t) + 0$$

d.h. $x(t)$ ist ein Vielfaches von $\vec{a}_1$



Die Daten akkumulieren sich um $\vec{a}_i$ herum.

Die $\vec{a}_i$ sind Zeilen von A.

$\vec{a}$ sind die Richtungen mit viel "Informationsgehalt"
(Wir denken da an PCA, Fisher usw.)

Eine Lösung des BSS Problems ist eine unmixing Matrix W und die Signale $y(t)$, die die Quellen approximieren.

$$y(t) = W^T x(t)$$

$$y(t) = \begin{pmatrix} a_1 s_1(t) \\ a_2 s_2(t) \end{pmatrix}$$

(Wir denken uns W so ein wenig als A^{-1})

Der Ansatz um das BSS Problem zu lösen, ist folgendes:

Wir nehmen an, dass die Quellsignale eine Eigenschaft P erfüllen. Diese Eigenschaft lässt sich numerisch mit einer Qualitätsfunktion q messen.
Dann maximieren wir $q(y(t)) = q(W^T x(t))$.

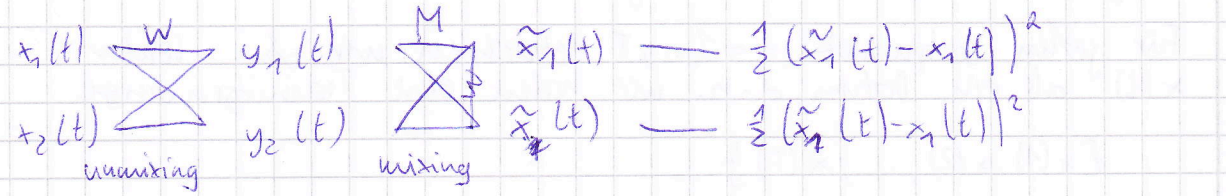
$$W^* = \arg\max_W q(W^T x(t))$$

$\tilde{y}_i(t) = \theta_{y_i}(t_1) + \dots + \theta_{k_i} y_i(t_k)$ die t_i sind dabei Zeitpunkte in der Vergangenheit

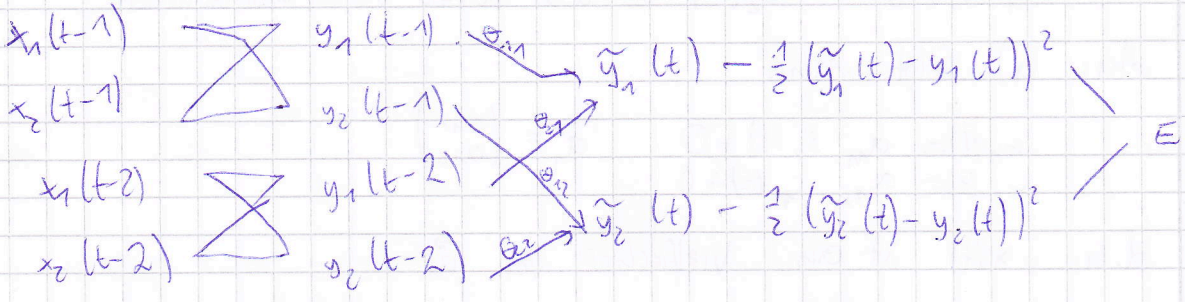
$(\tilde{y}_i(t) - y_i(t))^2 \cdot \frac{1}{2}$ = Fehler

je kleiner der Fehler, desto linearer ist $y_i(t)$

Rekurrente Netze



$$y(t) = W^T x(t)$$
$$\tilde{x}(t) = M^T y(t)$$
$$W^{-1} = M$$



Non-negative matrix factorization

Vorlesung 7.1.2009

- Wir haben unsere Datenbank V und wollen daraus die besten Merkmale (Basis) W und den Code H ermitteln, aus denen V am besten rekonstruiert werden kann

$$V \approx V' = W \cdot H$$

- Das Verfahren heißt nicht negativ, da man negative Komponenten vermeiden. Faktorisierung heißt gerade das Produkt W·H aus V zu finden

Der Algorithmus funktioniert wieder mit Expectation-Maximization und zwar ändern wir W und H immer abwechselnd, bis wir gut genug sind.
Zum Abbruch zweier nacheinanderfolgender Iterationen kann man entweder über den normalen euklidischen Abstand oder über Kullback-.

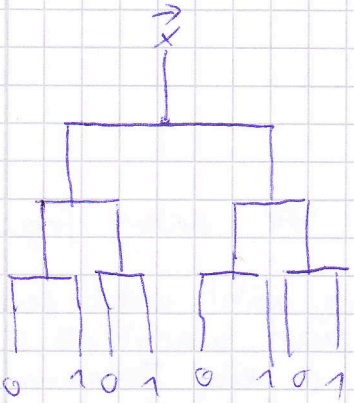
1. $W_{ia} \leftarrow W_{ia} \sum_\mu \frac{V_{i\mu}}{V'_{i\mu}} h_{a\mu}$ Korrigieren — Der 1. Index ist die Zeile im Merkmal und der 2. gibt das Merkmal an

2. $W_{ia} \leftarrow \frac{W_{ia}}{\sum_j W_{ja}}$ Normieren

3. $h_{a\mu} \leftarrow h_{a\mu} \sum_i W_{ia} \frac{V_{i\mu}}{V'_{i\mu}}$ Code korrigieren

Vorlesung 12
14.1.2009

ENTSCHEIDUNGSBÄUME



So was bietet sich an, wenn man mehrere Dimensionen hat

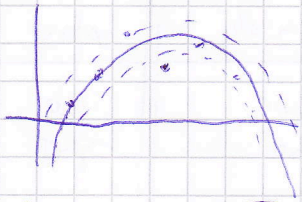
Man kann hier dann z.B. so fragen, ob im 1. Zweig $x_1 > 0{,}5$ ist und man je nach dem einen Ast nimmt. Dann kann man in der nächsten Ebene über x_2 fragen usw.

Jetzt ist die Frage, wie man die Reihenfolge der Abfrage geschickt wählt.

Bagging

Bootstrap aggregation.

Die Idee ist folgende:



Wir wollen die beste Anpassung an die Punkte haben, am besten ohne Ausreißer. Dann wollen wir ein Konfidenzintervall, in das z.B. 95% der Punkte reinfallen.

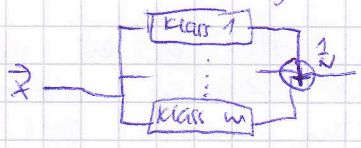
Trainingsmenge $T = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_N\}$

m Mal { Die Bootstrap Trainingsmenge funktioniert so, dass man N-mal zufällig ein Element selektiert und ~~Ersetzung~~ wieder zurücklegt.
Dann wird das Polynom ausgelesen

Das gibt uns m Funktionen, die uns den Durchschnitt von diesen m liefern, die wir dann nehmen.

Das praktische ist, dass sich das Konfidenzintervall automatisch aus allen Funktionen ergibt.

Der Entscheidungsbaum sieht so aus



Random Forests

Hier hat man N Punkte, K Merkmale, und die Merkmale haben Fehler. D.h. wir können damit umgehen, dass Merkmale bei Beispielen fehlen.

NW 2 Klassen

Die Idee ist so ein wenig wie bei Fisher. Wir haben folgendes:

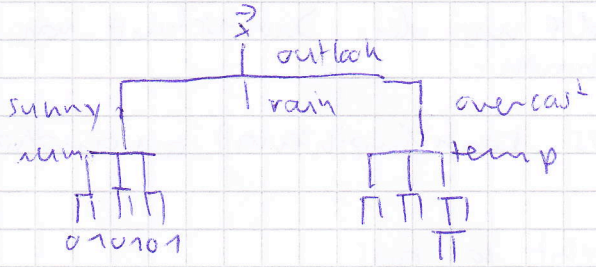
Für jede Ebene wählt man verschiedene Merkmale. Man macht das so lange, bis man mal so eine Trennung erhält, wo auf einer Seite des Klassifikators nur noch ein Punkt liegt. Da treffen wir dann tatsächlich die Entscheidung.

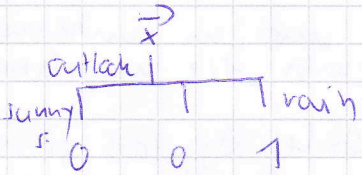
Die Höhe des Baums hängt von der Trainingsmenge ab, wie gut sie sich trennen lässt.

ID3 (Inductive - Decision - Tree)

Outlook	Temperature	humidity	windy	play (Klasse)
sunny	cold	85	yes	no
overcast	hot	70	no	yes
rain	temperate	50		
⋮	⋮	⋮	⋮	⋮

Aus dieser Tabelle von Daten über Baseballspiele kann man nun einen Baum machen, der Prognosen abgeben soll, ob bei bestimmten Bedingungen gespielt wird oder nicht.





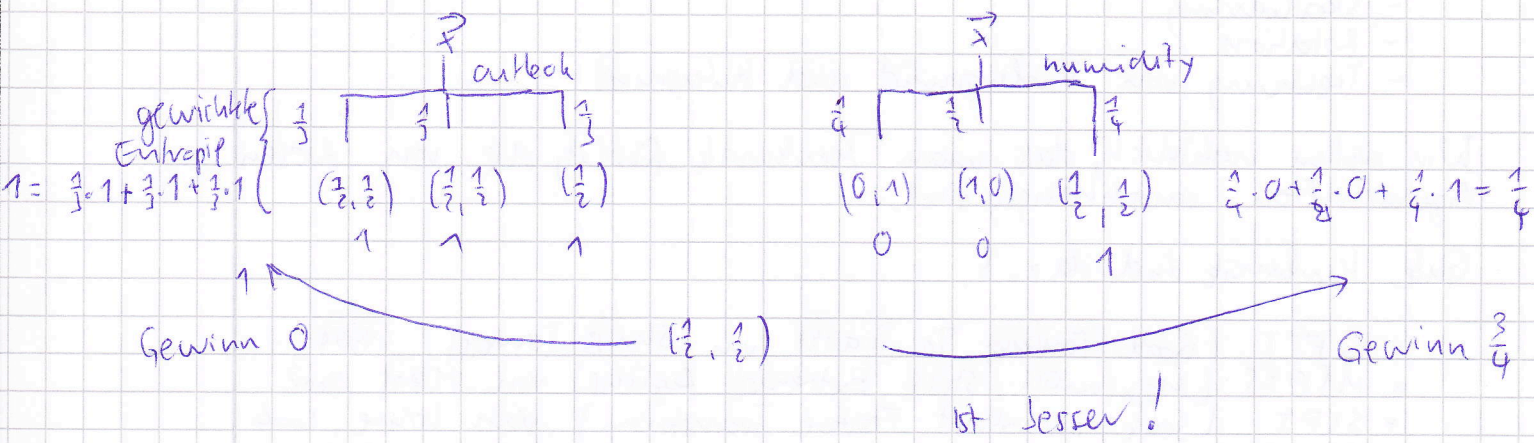
Welches Merkmal sollte man nun als erstes im Baum benutzen. Da ist es geschickt das zu nehmen, welches die Trainingsmenge am besten trennt. Das Maß, was ID3 verwendet ist dafür die Entropie der Trainingsmenge.

Die Entropie misst, wie viel Information man bekommen weiß.

Ja nein	(1,0)	Null Information
A B	(0,5, 0,5)	1 Bit
A B C D	(0,25, 0,25, 0,25, 0,25)	2 Bit

$T = (p_1, p_2, \dots, p_N)$ (Trainingsmenge nach Verteilung)

$E(T) = -\sum_{i=1}^{N} p_i \log_2 p_i$ (Entropie)



Man muss natürlich alle Paare von Merkmalen gegeneinander ausrechnen lassen.

Pseudocode für ID3:

R = Menge der Merkmale
C = Attribute
S = Trainingsmenge

if ($\forall \vec{x} \in S$, $\vec{x}$ von einer Klasse A)
  return Knoten mit Klasse A

if ($R = \emptyset$)
  return Knoten mit häufigster Klasse in S

Nehme Merkmal D mit maximalem Gewinn
return Knoten mit Markierung D und d Ausgängen (für jedes Attribut von D einen) und d Kindern (ID3 baum)
ID3 $(R - \{D\}, C, S_1), \dots, $ ID3 $(R - \{D\}, C, S_m)$
↑ Elemente für auf $D = d_1$ ↑ $D = d_m$

Der maximale Gewinn ergibt sich so:

Gewinn = Info(T) − Info(X, T)
↑ Merkmal

$\text{Info}(X,T) = \sum_{i=1}^{i} \frac{|T_i|}{|T|} \text{Info}(T_i)$ (Anzahl der Attribute des Merkmals X

Die Info ist die Entropie.

Eine Erweiterung von ID3 ist C4.5.

Merkmale in Bildern extrahieren

Vorlesung 1
21.1.2009

- Helligkeitsunterschiede
- Farben unterschiedlich
- Translation
- Skalierung
- Rotation
- Trennung von Vordergrund und Hintergrund

Wir wollen natürlich, dass man Merkmale findet, die von diesen Eigenschaften unabhängig sind.

Gute Merkmale sind da z. B.

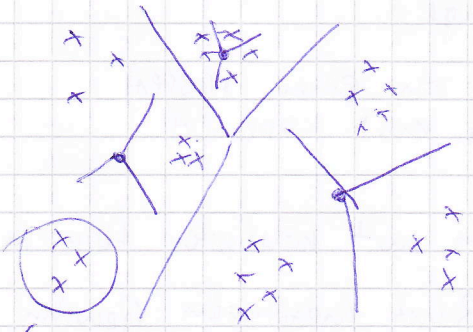
- GFTT (Good Features To Track) von Shi & Tomasi, 1994
- MSERS (Maximals, Stable Extremal Regions) von Matas, 2002
- SIFT (Scale invariant Feature Transform) von Lowe, 2004
- PCA-SIFT von Ke, 2004
- SURF (Speeded-Up Robust Feature) von Bay, 2006

Angenommen, man findet nun also solche guten Merkmale, hat aber für mehrere Objekte unterschiedliche Merkmalsdimensionen. Alles, was wir bisher hatten, funktioniert so nicht.

David Nister hat das 2006 wie folgt gelöst:

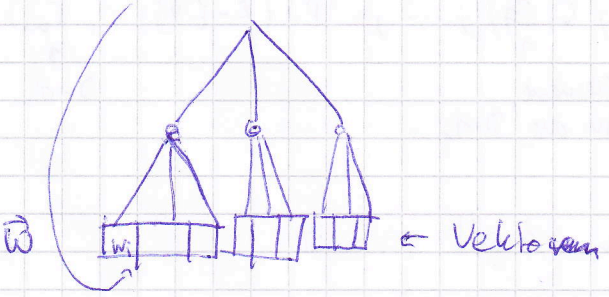
N Objekte

$\vec{m}_1, \ldots, \vec{m}_{o_i}$



k-means mit kleinem k

Daraus bekommt man rekursiv einen Baum



Die einzelnen Zellen kann man mit Gewichten $w_i = \ln\left(\frac{N}{N_i}\right)$ beschriften. Das ist das Gütekriterium für Merkmale.

$d_i = w_i \cdot n_i$ mit n_i ist die Anzahl der Merkmale bei Index i

Das ist der "Fingerabdruck".

Für jedes Objekt in der Datenbank wird ein Fingerprint $\vec{d}_i$ erstellt.

Für ein Anfrageobjekt wird ebenfalls solch ein Fingerprint $\vec{q}$ erstellt

$\vec{q_i} = w_i$ ~~wenn~~ n_i

Als Ähnlichkeitsmaß nehmen wir ein s.

$$s(\vec{d}, \vec{q}) = \left\| \frac{\vec{q}}{\|\vec{q}\|} - \frac{\vec{d}}{\|\vec{d}\|} \right\|$$

Das macht man für alle Objekte der Datenbank, d.h. wir wollen

$$\operatorname{argmin}_i s(\vec{d_i}, \vec{q})$$

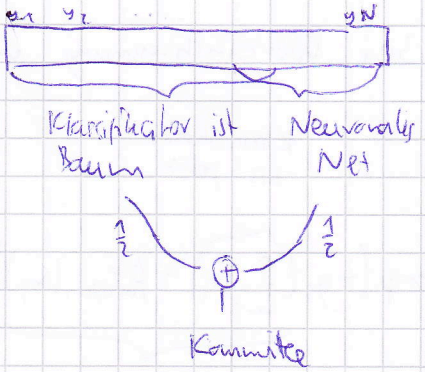
Als Optimierung kann man auch $\operatorname{argmax}_i \langle \vec{q}, \vec{d_i} \rangle$ berechnen.

Vorlesung 14
28.1.2009

Boosting

Weak classifiers. Wir haben ein "Komitee" von Klassifikatoren $f_1, f_2, \dots, f_m$, wobei jedes $f_i(x) \in \{-1, 1\}$ ist.
Das Komitee ist dann $f(x) = \sum_{i=1}^{m} \alpha_i f_i(x)$ mit $\alpha_i = \frac{1}{m}$

In der Praxis kann man z.B. das so machen:



Der Ansatz für das Boosting ist nun, dass wir einen Gesamtklassifikator haben möchten.

$$G(x) = \operatorname{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x) \right)$$

$G(x)$: 1 stark

$G_m(x)$: schwacher Klassifikator (Stümpf) mit Trefferquote > 50%

Der Algorithmus, den wir uns anschauen, heißt AdaBoost.

Dabei haben wir eine Trainingsmenge

$\{(x_1, y_1), (x_2, y_2) \dots\}$, die gewichtet ist durch $\{w_1, w_2, \dots\}$

1. Gewichte setzen auf $w_i = \frac{1}{N}$ für $i = 1, \dots, N$
2. For $m = 1$ to M (das sind die schwachen Klassifikatoren)
   a) Trainiere schwachen Klassifikator G_m mit Trainingsmenge T und den Gewichten w_i, $i = 1, \dots, N$

b)
$$err_m = \frac{\sum_{i}^{N} w_i \, I(y_i \neq G_m(x_i))}{\sum_{i}^{N} w_i}$$

I ist dabei eine Indexvariable, die 1 ist, wenn die Bedingung zutrifft, 0 sonst.

c) $\alpha_m = \log \frac{1 - err_m}{err_m}$

d) $w_i \leftarrow w_i \cdot \exp\left(\alpha_m I(y_i \neq G_m(x_i))\right)$ für $i = 1, \dots, N$

$(w_i^{(m+1)} = w_i^{(m)} \cdot \dots)$

3) $G(x) = \text{sign}\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right)$

Wir wollen natürlich zeigen, dass das auch gut geht und stimmt.

Fehlerfunktion: $L(y, f(x)) = \exp(-y \cdot f(x))$

$\{-1,1\}$ $\{-1,1\}$

die Fehlerfunktion gibt also e^{-1} bei Übereinstimmung und e^{1} sonst aus.

Wir wollen folgendes lösen:

(Klassifikator bis jetzt)

$$(\beta_m, G_m) = \arg\min_{\beta_m, G_m} \left(\sum_{i=1}^{N} \exp\left(-y_i \underbrace{\left(f_{m-1}(x_i) + \beta_m G_m(x_i)\right)}_{\text{neuer Klassifikator}}\right)\right)$$

$$\Rightarrow (\beta_m, G_m) = \arg\min_{\beta_m, G_m} \left(\sum_{i=1}^{N} w_i^{(m)} \exp(-\beta_m y_i G_m(x_i))\right) \quad \text{mit} \quad w_i^{(m)} = \exp(-y_i f_{m-1}(x_i))$$

Zunächst suchen wir nun das beste G_m. Wir behaupten, dass

$$G_m = \arg\min_{G_m} \sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i))$$

$$(\beta_m, G_m) = \arg\min_{\beta_m, G_m} \left(e^{-\beta_m} \sum_{y_i = G(x_i)} w_i^{(m)} + e^{\beta_m} \sum_{y_i \neq G(x_i)} w_i^{(m)} \right)$$

$$= \arg\min_{\beta_m, G_m} \left(\left(e^{\beta_m} - e^{-\beta_m}\right) \sum_{i=1}^{N} w_i^{(m)} \underbrace{I(y_i \neq G(x_i))}_{\lambda} + e^{-\beta_m} \sum_{i=1}^{N} w_i^{(m)} \right)$$

Wir schlussfolgern, dass wir $\sum w_i^{(m)} I(y_i \neq G_m(x_i))$ minimieren.
Dann finden wir das optimale β für

$$\left(e^{\beta} - e^{-\beta}\right) \underbrace{\sum_{i=1}^{N} w_i^{(m)} I(y_i \neq G(x_i))}_{A} + e^{-\beta} \underbrace{\sum_{i=1}^{N} w_i^{(m)}}_{B}$$

Dann sehen wir $(e^{\beta} + e^{-\beta}) A - e^{-\beta} B = 0$

$(e^{2\beta} + 1) A - B = 0$

$e^{2\beta} \cdot A = B - A$

$e^{2\beta} = \frac{B-A}{A}$

$\beta = \frac{1}{2} \log \frac{B/B - A/B}{\frac{A}{B}}$

$= \frac{1}{2} \log \left(\frac{1 - err_m}{err_m} \right)$

$\frac{A}{B} = \frac{\sum w_i^{(m)} I(y_i \neq G(x_i))}{\sum w_i^{(m)}} = err_m$

$w_i^{(m+1)} = w_i^{(m)} \cdot e^{-\beta_m y_i G_m(x_i)}$

$- y_i G_m(x_i) = 2 (I(y_i \neq G_m(x_i)) - 1$

$w_i^{(m+1)} = w_i^{(m)} \cdot e^{\alpha_m I(y_i \neq G_m(x_i))} \cdot \underbrace{e^{-\beta_m}}_{\text{Multiplikative Konstante für die ganze Trainingsmenge}}$

In AdaBoost lässt man diese Konstante einfach weg.