panikzettel.htwr-aachen.de

Elements of Machine Learning and Data Science

Version 29 | 04.03.2024

Jonas Schneider Adrian Groh

Inhaltsverzeichnis

1 Einleitung und Stochastisches Vorwissen

Vorwort

Dieser Panikzettel ist über die neue Vorlesung „Elements of Machine Learning and Data Science“, ein Pflichtfach des neuen Informatik Bachelors der RWTH Aachen.

Dieser Panikzettel ist Open Source auf `https://git.rwth-aachen.de/jonas.max.schneider/panikzettel` . Wir freuen uns über Anmerkungen und Verbessungsvorschläge (auch von offiziellen Quellen).

Es ist häufig schwierig, die genauen Themenschwerpunkte der Vorlesung und Prüfung zu erkennen. Hier allerdings besonders, da es die erste Iteration des neuen Fachs ist und keine Altklausuren zu Rate stehen. Wir werden deshalb einige Zeit (und Feedback von späteren Semestern) brauchen um besser Zusammenfassen zu können.

Da das neue Fach auf Englisch gehalten wird, ist es eine gute Zeit, dies auch für den Panikzettel umzusetzen und eine zusätzliche Englische Version zu erstellen. Dies hier ist die **deutsche** Version, jedoch verwenden wir die englischen Fachwörter um keine Übersetzungsfehler bei uns und euch zu riskieren.

Stochastisches Vorwissen

Ein großer Teil der Machine Learning und Data Science Konzepte und Algorithmen beruhen auf Stochastik. Wir werden hier *nicht* die Grundlagen der Stochastik wiederholen, auch wenn ein kleiner Teil der Vorlesung genau dies tat, sondern auf den eigenen Stochastik Panikzettel[1] verweisen.

Aufteilung

Die Vorlesung wurde von drei verschiedenen Lehrstühlen und Professoren zusammen angeboten und aufgeteilt. Die Aufteilung dieses Panikzettel orientiert sich ebenfalls an dieser Aufteilung.

- Machine Learning
- Data Science
- Evaluation und AutoML

2 Machine Learning

Das Grundkonzept des gesamten Machine Learning Bereichs, der in dieser Vorlesung behandelt wird, kümmert sich um *Klassifizierung*, also der Aufteilung in Klassen (z.B. gegeben ein Foto, ist es eine Katze oder ein Hund).

Lernformen & Lernziele

Machine Learning Algorithmen lernen mit Trainingsdaten. Diese können bereits (teilweise) klassifiziert sein (d.h.*labeled*), woraus sich somit drei Arten zu lernen ergeben:

- Supervised learning
- Semi-supervised learning
- Unsupervised learning

Wir werden einige Lernziele besprechen, aber es ist gut, sie vorher eingeführt zu haben.

[1] panikzettel.htwr-aachen.de

	Discrete Targets	Continuous targets
Known Targets / Supervised Learning	*Classification* Einteilung in verschiedene (zuvor bekannte) Klassen	Regression Versuchen mittels Funktionen (z.B. lineare) Daten zu beschreiben.
Unknown Targets / Unsupervised Learning	Clustering Einteilung unbekannter Datenpunkte in *Clusters*, die stärker untereinander als mit anderen Verbunden sind.	Density estimation Versuchen, eine Wahrscheinlichkeitsverteilung abzubilden von zufälligen samples. Anders als bei Regression ist hier die Datenfunktion nicht bekannt / es wurde nicht vorher gelabeled.

Das Ziel wird aber immer sein: Gegeben Trainingsdaten $\mathcal{D} = \{x_1, ..., x_n\}$ oder mit *labels* $\mathcal{D} = \{(x_1, t_1), ..., (x_n, t_1)\}$ eine Funktion y zu trainieren, die zu Daten, die nicht im Testset vorkommen, eine Vorhersage treffen kann.

2.2 Probability Density Estimation

Wie wahrscheinlich ist es, dass wir ein x sehen unter der Bedingung, dass wir in der Klasse $\mathcal{C}_k$ sind, also $p(x \mid \mathcal{C}_k)$. Diese *likelihood* ist genau das Gegenteil von dem was wir brauchen: $p(\mathcal{C}_k \mid x)$.

Somit stellen wir es mit den *a-priori probabilities* $p(\mathcal{C}_k)$ und Bayes Theorem um:

$$p(\mathcal{C}_k \mid x) = \frac{p(x \mid \mathcal{C}_k)p(\mathcal{C}_k)}{p(x)} = \frac{p(x \mid \mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(x \mid \mathcal{C}_j)p(\mathcal{C}_j)}$$

Dies ist die *posterior* Wahrscheinlichkeit, die wir benötigen, um x zu klassifizieren.

Nun verteilt sich $p(\mathcal{C}_k \mid x)$ in einer bestimmen Wahrscheinlichkeitsverteilung, die wir aber nicht kennen, die es nun geht zu approximieren.

Gegeben ist eine Reihe von Klassen $\mathcal{C}_1, ...\mathcal{C}_d, d \in \mathbb{N}$ (oder $\mathcal{C}_a, \mathcal{C}_b$), deren *a-priori* Wahrscheinlichkeiten und ein Trainingsdatensatz $x_1, ..., x_N$, $N \in \mathbb{N}$ mit Labeln.

2.3 Parametrische Methoden mittels Gaussian (Normal) Verteilung

Wir nehmen nun an, dass die $p(\mathcal{C}_k \mid x)$ nach einer Normal Verteilung $\theta_N = (\mu, \sigma)$ (genauer irgendeiner Verteilung mit Parametern θ) verteilt ist. Um hier die Parameter zu bestimmen, benötigen wir erneut die Hilfe von der log-likelihood und der Maximum Likelihood. Wir werden das hier nochmal durchgehen:

Hierfür benötigen wir die Wahrscheinlichkeit $L(\theta) = p(\mathcal{X} \mid \theta)$, dass $\mathcal{X}$ von den Parametern θ generiert wurde. Wir wollen $L(\theta)$ maximieren. Für einen einzelnen Datenpunk x_n gilt $p(x_n \mid \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x_n-\mu)^2}{2\sigma^2}\right)$.

Da alle Variablen i.i.d. (d.h. unabhängig und identisch verteilt) sind, können wir die Wahrscheinlichkeiten summieren: $L(\theta) = \sum_{n=1}^{N} p(x_n \mid \theta)$ und das *negative log-likelihood* bilden $E(\theta) = -\ln L(\theta) = -\sum_1^n \ln p(x_n \mid \theta)$ ($\triangle \ln a * b = \ln a + \ln b$).

Um nun das Maximum zu berechnen ist nun die Ableitung von $E(\theta)$ erforderlich:

$$\frac{\partial}{\partial\theta}E(\theta) = -\sum_{n=1}^{N}\left(\frac{1}{p(x_n \mid \theta)}\right)\cdot\frac{\partial}{\partial\theta}p(x_n \mid \theta)$$
$$= \ldots = \frac{1}{\sigma^2}\sum_{n=1}^{N}x_n - N\mu \stackrel{!}{=} 0$$
$$\iff \hat{\mu} = \frac{1}{N}\sum_{n=1}^{N}x_n \wedge \hat{\sigma}^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \hat{\mu})^2$$

.

Bei Maximum Log-Likelihood Optimierungen wird die Varianz unterschätzt, und muss im Fall der Normalverteilung mit $\frac{N}{N-1}$ ausgeglichen werden.

2.4 Non-Parametrische Methoden: Histogramme, Kernel Methods & k-Nearest Neighbors

Das hier sind Methoden, die nicht die Parameter der *probability density* Funktion (pdf) approximieren wollen, sondern Die Funktion als Ganzes. Wir wissen also gar nicht die unterliegende pdf und somit auch nicht ihre Parameter.

Histogramme

Aufteilung der Daten in N Säulen mit Breite Δ. Die Höhe der Säule ist dann $p_i = \frac{n_i}{N\Delta_i}$, wobei n_i die Anzahl der Beobachteten Ergebnisse in dem Bereich i ist.

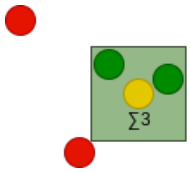

Abbildung 1: Kernel Method

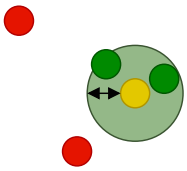

Abbildung 2: 1-Nearest Neighbors

Gegeben sei eine *probability density function* (pdf) $p(\mathcal{C}_k \mid x)$, die wir erneut approximieren wollen. Wir visualisieren hier mit zwei Dimensionen. Wir können nun p approximieren indem wir uns einen kleinen Bereich $\mathcal{R}$ um die Datenpunkte angucken und somit ein $P = \int_{\mathcal{R}} p(y)dy = \frac{K}{N} \approx p(x)V$ erhalten, wobei V das Volumen von $\mathcal{R}$ ist und K die Anzahl der Datenpunkte die in $\mathcal{R}$ liegen.

Wir haben somit zwei Möglichkeiten zu approximieren:
Kernel Methods Fixes Volumen V, aber mit einer variablen Menge von Punkten K in $\mathcal{R}$
K-Nearest Neighbors Fixe Punktzahl K in $\mathcal{R}$, somit unterschiedlich große Volumen V

Kernel Methods

Wir bilden einen Kernel k mit $k(u) \geq 0$ und $\int k(u)du = 1$ Hierbei beschreibt u einen Vektor von einem Punkt x zu einem x_n. Das k gewichtet diese Distanz dann.

Somit gilt $K = \sum_{n=1}^{N} k(x - x_n)$ und $p(\mathcal{C}_k \mid x) = \frac{K}{NV} = \frac{1}{N}\sum_{n=1}^{N} k(x - x_n)$

Der parzen-window (kernel), indem man einen Würfel der Höhe $h \in \mathbb{N}$ um x aufbaut, wäre als Beispiel

$$k = \begin{cases} \frac{1}{h^D} \text{ if } |u_i| \leq \frac{1}{2}h, i = 1, \ldots D \\ 0 \text{ else} \end{cases}$$

Also skaliert 1, falls x_n (in allen Dimensionen) im Würfel liegt und 0 sonst.

Somit ergibt sich eine probability density estimation von $p(\mathcal{C}_k \mid x) \approx \frac{K}{NV} = \frac{1}{Nh^D}\sum_{n=1}^{N} k(x - x_n)$

Populäre kernels sind z.B. auch Gaussian Blurs (Smoothener)

k-Nearest Neighbors
Hier setzten wir ein $K \in \mathbb{N}$ fest und bestimmen eine Sphäre, die wir brauchen, um die K nächsten Nachbarn zu inkludieren. Somit gilt immer noch $p(x) \approx \frac{K}{NV}$.

Die allermeisten *non-parameterized* probability density estimators haben bestimmte Einstellungen, die für unterschiedliche Ergebnisse sorgt, sogenannte *Hyperparameter* (z.B. Δ_i, h, K, etc.).

k-NNs können mittels Bayes Decision Theory auch als Klassifizierer dienen. Indem man die class-conditional Verteilung approximiert mit $p(x \mid C_j) \approx \frac{K_j}{N_j V}$ und auch durch die prior$p(C_j) \approx \frac{N_j}{N}$ die posteriors $p(C_j \mid x) \approx p(x \mid C_j)p(C_j)\frac{1}{p(x)} = \frac{K_j}{K}$

2.5 Mixture Models

In einem Mixture Model wird davon ausgegangen, nach mehreren ($M \in \mathbb{N}$) probability distributions verteilt zu sein, die sich addieren. Meist wird die Normalverteilung gewählt.

Also $p(x \mid \theta) = \sum_{j=1}^{M} p(x|\theta_j)p(j)$, wobei $\theta = (\pi_1, \theta_1, ..., \pi_1, \theta_M)$ alle Parameter bündelt und $p(j) = \pi_j$ die priors (also die Anfangsvermutungen des *mixtures components*).

Um hieraus eine formale Wahrscheinlichkeitsverteilung zu erstellen, muss $\sum_{j=1}^{M} \pi_j = 1$ und somit $\int p(x|\theta)dx = 1$ sein.

Hierfür gibt es kein analytisches Verfahren, die Maximum Log Likelihoods zu finden. Somit bleiben numerische Verfahren (und) iterative Optimierungsverfahren.

EM Algorithmus
Der *E-STEP*:

$$\gamma_{j(x_n)} \leftarrow \frac{\pi_j \mathcal{N}(x_n \mid \mu_j, \Sigma_j)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n \mid \mu_k, \Sigma_k)}$$

Hier ist Σ die Kovarianzmatrix.

Der *M-STEP*:

$$\hat{N} \leftarrow \sum_{n=1}^{N} \gamma_{j(x_n)} \qquad \hat{\mu}_j \leftarrow \frac{1}{\hat{N}_j} \sum_{n=1}^{N} \gamma_{j(x_n)} x_n$$

$$\hat{\pi}_j \leftarrow \frac{\hat{N}_j}{N} \qquad \hat{\Sigma}_j \leftarrow \frac{1}{\hat{N}_j} \sum_{n=1}^{N} \gamma_{j(x_n)} (x_n - \hat{\mu}_j)(x_n - \hat{\mu}_j)^T$$

Der EM Algorithmus muss durch *regularization* gegen $\sigma \to 0$ geschützt werden, weswegen $\sigma_{\min} I$ addiert wird[2]

2.6 Linear Discriminants

Linear Discriminants versuchen, ein Gerade $y(x) = w^T x + w_0$ zu finden, die ein Datensatz trennt (d.h. $y(x) \geq 0 \implies C_1$ sonst C_2). Falls dies gelingt, heißt ein Datensatz *linearly separable*.

Häufiger wird aber $y(x) = \tilde{w}^T \tilde{x} = \sum_{i=0}^{D} w_i x_i$ benutzt, wobei hier $x_0 = 1$ gesetzt ist und der Bias w_0 somit verrechnet wird.

[2]Der EM Algorithmus wird aus meiner Sicht nicht groß in der Klausur angewandt werden können, Wissensfragen jedoch schon.

Es lassen sich $K \in \mathbb{N}$ *linear discriminants* berechnen und die Klasse C_k genau dann gewählt werden, wenn $y_k(x) > y_j(x)$ für alle $j \neq k$.

Nun aber zur Optimierungsmethoden dieser $y_k(x)$. Zunächst werden alle k Diskriminanten zusammen gruppiert $\widetilde{W} = (\tilde{w}_1, ..., \tilde{w}_K) = \begin{pmatrix} w_{10} & \dots & w_{K0} \\ \vdots & \ddots & \vdots \\ w_{1D} & \dots & w_{KD} \end{pmatrix}$, $\tilde{X} = \begin{pmatrix} x_1^T \\ \vdots \\ x_N^T \end{pmatrix}$ und $Y(\tilde{X}) = \tilde{X}\widetilde{W}$. Ebenso werden die *target vectors* (die label) $T = \begin{pmatrix} t_1^T \\ \vdots \\ t_N^T \end{pmatrix}$.

Um lernen zu können, definieren wir eine *error function* (hier *Sum of squares*) $E(W) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{K}\left(w_k^T x_n - t_{nk}\right)^2$. Die $\frac{1}{2}$ als Faktor ist nicht benötigt, mach aber bei der Differenzierung den Faktor 2 weg. Nehmen wir nun im 2-class Fall die Ableitung: $\frac{\partial E(w)}{\partial w} = ... = w = \left(X^T X\right)^{-1} X^T t = X^\dagger t$ (das $X^\dagger$ ist die pseudo-inverse, da X ja singulär seine könnte). Somit erhalten wir eine *closed-form* Lösung $y(x; w) = w^T x = t^T \left(X^\dagger\right)^T x$.

Um weniger sensitiv gegen *outliers* zu sein, wird y allerdings meist mit einer Aktivierungsfunktion versehen (ähnlich wie bei NNs) $y(x) = g(w^T x)$. In der Vorlesung der Logistic-Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$.

Basis functions ϕ erweitern die Gerade zu $y(x) = w^T \phi(x)$, um nicht linear trennbare Datensätze zu klassifizieren zu können.

2.7 Regressions

Wir haben zwei verschiedene Arten Regression behandelt:

Linear Regression approximieren einer Funktion $h(x) \in \mathbb{R}$, gegeben durch label $t_n = h(x) + \varepsilon$

Logistic Regression approximieren einer diskreten Funktion (bei Linearer Regression sind zwar auch nur Datenpunkte gegeben, aber hier ist die Funktion diskret)

Linear Regression

Für linear regression wenden wir die Least Square Regression an. Wir nehmen als *error function* erneut die Sum of squares Funktion $E(w) = \frac{1}{2}\sum_{i=1}^{N}\left(y(x_n; w) - t_n\right)^2$.

Kommen also bei dem gleichen $w = \left(\Phi^T \Phi\right)^{-1} \Phi^T$ an wie bei den Diskriminanten. Allerdings mit $y(x) = w^T \phi(x)$ einer Basis Funktion

Nehmen wir also als Basis Funktion $\phi_{j(x)} = x^j$. Nun ist die Wahl des Polynomgrads ein wichtiger *Hyperparameter*. Jedoch ist mit dieser Error Funktion das *overfitting* Risiko groß, da es den Datenpunkten unterliegenden *noise* modelliert und nicht $h(x)$.

Um dagegen vorzugehen, wird ein *regularizer* Ω (z.B. $\Omega = \frac{1}{2}\|w\|^2$) eingesetzt $E(X) = L(w) + \lambda\Omega(w)$, hierbei ist $L(w)$ der Loss-Term also einfach die vorherige *error function*.

Logistic Regression

Wir wollen die Class-posteriors $p(\mathcal{C}_1 \mid \phi)$ modellieren und zwar als *linear discriminant* $y(\phi) = \sigma\left(\mathcal{C}_1^T \phi\right)$. Wir gehen also nicht der generativen Modellierung nach (die direkt die posterior Wahrscheinlichkeitsverteilung modellieren will), sondern **nur** der Grenze zwischen den Klassen (*discriminative modelling*) [Note: hierfür sind labels zwangsweise nötig].

Falls ihr gefragt werdet, was die Vorteile sind:

- Effizientere Parameter Nutzung, da weniger Parameter gebraucht werden.

Zudem gibt es noch zwei wichtige Fachwörter:

cross-entropy error $E(w) = -\sum_{n=1}^{N}(t_n \ln y_n + (1 - t_n)\ln(1 - y_n))$

Softmax Regression $\frac{\exp(a_k)}{\sum_{j=1}^{K}\exp(a_j)}$ für eine Klasse $k \in \underline{K}$. (Die beiden Sachen lassen sich auch verbinden :/)

Für die tatsächliche Regression müssen wir wieder auf iterative Methoden zurückgreifen. In diesem Fall auf *gradient descent*.

Gradient Descent

$$w_{kj}^{\tau+1} = w_{kj}^{\tau} - \eta \frac{\partial E(W)}{\partial w_{kj}}\bigg|_{w^{\tau}} \quad (\text{fortan } w^{\tau+1} = w^{\tau} - \eta \Delta E(w))$$

dies ist die (so halb) erste Taylor expansion mit Hyperparameter *learning rate* η. Ist die *learning rate* zu hoch, wird das Optimum „übersprungen“ und es kann sein, dass unsere Funktion *divergent* wird, also sich immer zwischen zu viel Error auf der einen + Seite zu zu viel auf der - Seite (vereinfach). Bei zu kleiner Learning Rate dauert es lange und eventuell konvergiert der Gradient zu einem lokalen Minimum statt zu einem Globalen.

Die Newton-Raphson Methode (orientiert sich an der zweiten Taylorentwicklung) verwendet zusätzlich noch ein $H^{-1} = \frac{\partial^2 E(w)}{\partial w_i \partial w_j}$

Support Vector Machines (SVM)

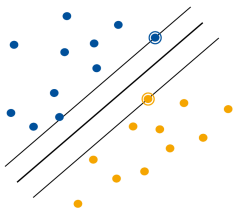

Abbildung 3: Example of a SVM

Bei Support Vector Machines (SVMs) wird anstatt eine Diskriminante zu verwenden, versucht, zwischen den zwei Cluster eine Safety Zone (d.h. *margin*) aufzubauen und zu maximieren. Das $y(x) = (t_n(w^T x_n + b))$, welches den größten *margin* $\frac{1}{2}\|w\|^2$ erschafft und $y(x) \geq 1$ für alle $n \in N$ (d.h. alle Trainingspunkte richtig klassifiziert), gilt als optimiert.

Hieraus bauen wir uns ein Optimierungsproblem à la Quantitative Methoden (BWL☒). Wie genau wir dahin kommen, ist für den Panikzettel (und aus meiner Sicht) wenig relevant, doch brauchen wir hierfür *lagrange multiplier* λ, a_n für die Primal form.

Primal Form

$$L(w,b,a) = \frac{1}{2}\|w\|^2 - \sum_{n=1}^{N} a_n \left[t_{n(w^T x_n + b)} - 1\right]$$

Conditions:

$$\begin{aligned} a_n &\geq 0 \\ t_{n(w^T x_n + b)} - 1 &\geq 0 \\ a_n\left[t_{n(w^T x_n + b)} - 1\right] &= 0 \end{aligned}$$

Karush-Kuhn-Tucker conditions:

$$\lambda \geq 0$$
$$f(x) \geq 0$$
$$\lambda f(x) = 0$$

Die Intuition aus der Primal Form ist (Gut, dass man sagen muss was die Intuition ist), dass nur manche Datenpunkte (die support vectors (logisch)) die Margins und die Decision Boundary beeinflussen.

Zeit Komplexität: $O(D^3)$, also sehr schlecht für höherdimensionale Daten oder wenn man Basis functions benutzen möchte.

Dual Form

$L_{d(a)} = \sum_{n=1}^{N} a_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} a_n a_m t_n t_m (x^T x_n)$

mit conditions:

$$a_n \geq 0 \quad \forall n \in \underline{N}$$
$$\sum_{n=1}^{N} a_n t_n = 0$$

Zeit Komplexität: $O(N^3)$, also nichtmehr abhängig von D.

Zudem sind die meisten Datenpunkte $a_n = 0$, was also sparse matrix Zeugs ermöglicht (wer HPC kann, weiß, dass das gut zu haben ist). Ist natürlich immer noch anstrengend für große Datenmengen.

Was ist aber wenn die Daten nicht linear trennbar sind?

Dann lassen sich mit *Soft-Margin SVMs* die „falschen“ Punkte bestrafen, indem Slack Variablen $\varepsilon_n = |t_n - y(x_n)|$ für alle Trainingspunkte eingefügt werden. $\varepsilon_n = 0$ bei Korrektheit und erfährt bis 1 eine penalty.

Alle Slacks werden dann summiert und mit einem C, dem tradeoff Hyperparameter, gewichtet ($\frac{1}{2} \|w\|^2 + C \sum_{n=1}^{N} \varepsilon_n$). Die Dual Form ist mit Slack variablen deutlich günstiger. Somit sollte die Primal Form wenig Dimensionen, viele Datenpunkte und *linear separable* sein.

2.8 Error Funktion Analyse

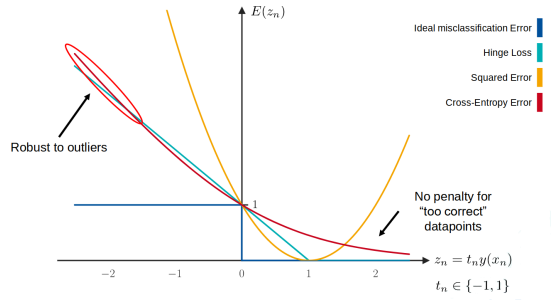

Abbildung 4: Error Funktionsanalyse

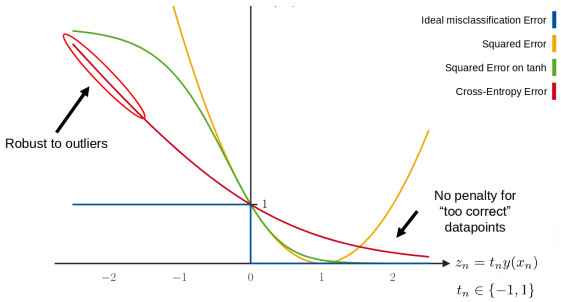

Abbildung 5: Weitere Error Funktionsanalyse

In Abbildung werden die verschiedenen Error Funktionen geplottet. Was sind hier die Achsen? Auf der Y-Achse ist der Fehler. Auf der X-Achse ist die Zuversicht die die Prediction hatte, bei 0 ist er sich unsicher, bei negativen Zahlen hat er Falsch gelegen, bei positiven Richtig.

- Die Optimale Fehlerfunktion ist die Step Funktion, allerdings hat die den Nachteil keinen Gradienten zu haben und somit ist *gradient descent* nicht möglich.
- Der Squared Error steigt bei negativen zu schnell, und steigt bei „sehr richtigen“ (also z.B. Punkte weit weg von der Boundary).
- Hinge Loss ist bei X=1 nicht differenzierbar.

- Cross-Entropy Error steigt im negativen Linear (gut für Gradient descent) und hat kein positives „penalty“
- Tanh hat zwar kein positives „penalty“, aber dafür genau wie die Step Funktion keine große Steigung im negativen Bereich.

2.9 Neural Nets

Ein *perceptron* bekommt eine Menge $D \in \mathbb{N}$ Inputs, gewichtet die dann und summiert sie. Das Ergebnis ist dann $y(x) = \sigma\Big(b + \sum_{i=1}^{D} w_i x_i\Big)$, b ist der Bias, σ ist die *activation function* (auch hier werden die Gewichte meistens in eine einheitliche Matrix gepackt, weswegen die Formel sich ändern kann). Perceptrons sind also nur generalisierte Lineare Discriminants.

Nun fügen wir mehrere Layers von Perceptrons zusammen und bilden daraus ein *Multi-Layer Perceptron*. Jeder Layer fügt einen *bias* (z.B 1) hinzu, der dann mit Weights zu jedem Perceptron der Layer verbunden ist.

Die Weights und Biases werden dann durch Backpropagation und Stochastic Gradient Descent trainiert. Auch hier ist die *learning rate* ein wichtiger Hyperparameter, genauso wie der Aufbau des NN selber.

3 Data Science

In Data Science sprechen wir über *features*

Name	Farbe	Kosten	Bestseller
Porsche GT	Red	5€	Yes
Fiat 500e	Blue	inf	No

Tabelle 1: Beispiel Labeled Data

In Tabelle Tabelle 1 gibt es die Features Name, Farbe, Kosten, Bestseller. Bestseller könnte hier ein *target-feature* sein, was wir vorhersehen möchten. Die Reihen sind die *instances*.

Die Datentypen sind ähnlich wie die der Deskriptiven Statistik. Zwei Sachen nur: Nominal sind wie Enums (ungeordnet). Ordinal sind z.B. Sternbewertungen.

Visualisierung Histogramme, Scatter Plots, Box Plots.
Correlation $\mathrm{Corr}(x, y) = \frac{\mathrm{Cov}(x,y)}{\sqrt{\mathrm{Var}(X)} \cdot \sqrt{\mathrm{Var}(y)}}$
Binning Aufteilen von *continuous features* zu *categorical features*.
Hier gibt es zwei Versionen:
- Equal Width Binning: z.B. 5. dann startet der erste Bin bei dem niedrigsten z.B. $[2, 7), [7, 12), \ldots$
- Equal Frequency Binning: z.B. 3 pro Bin. Ist logisch.

3.1 Decision Trees

Ein Decision Tree trennt die Gesamtheit in mehreren Schritten auf und versucht so zu „klassifizieren“ bzw. vorherzusagen. Das Target Label steht in den *leaf nodes*. Im generellen versucht man die target labels in den Leaf Nodes zu trennen (z.B. zwischen „Yes“ Bestseller oder „No“ Bestseller), aber dabei sollte der Baum so klein und unkompliziert wie möglich sein.

Hierfür brauchen wir ein paar Metriken (die sollte man wirklich können):
Entropy $H(t) = -\sum_{k=1}^{K} (p(t = k) \cdot \log_2 p(t = k))$. Man sollte hier auf die features und Wahrscheinlichkeiten aufpassen. Beispiel: 2 Blaue, 3 Rote Kugeln $H(\text{Farbe}) = -\big(\frac{2}{5} \log_2 \frac{2}{5} + \frac{3}{5} \log_2 \frac{3}{5}\big) \approx 0.97$.

Die Minimale Entropie wäre 5 Rote Kugeln $H(\text{Farbe}) = 0$ und Maximale ist immer die gleichverteilung von allen Möglichen $H(\text{Farbe}) = \log_2(K)$ wo $K \in \mathbb{N}$ die Anzahl der Kugeln ist.

Overall Entropy In einem Decision Tree berechnet die Overall Entropy indem die Anzahl der Sachen im *leaf nodes*. Gegebenenfalls betrachten wir nur splits von dem Feature d.
$H_W = \sum_{\text{node} \in \text{leaf\_nodes}(d)} \left(\frac{|\text{node}|}{N} \cdot H^{\text{node}}(t) \right)$

Entropy-Information Gain Ist einfach nur der Unterschied vom Start bis zu einem *leaf node*.
$\text{IG}(d) = H(t) - H^d_{W(t)}$

Entropy-Information Gain Ratio Hier nehmen wir
$\text{GR(d)} = \left(\frac{\text{IG}(d)}{H(d)} \right)$

Gini Index $\text{Gini}(t) = 1 - \sum_{k=1}^{K} p(t=k)^2$

Warum es die Entropy und den Information Gain gibt sollte klar sein, aber warum brauchen wir den GR und Gini?

- Information Gain Ratio bestraft feature splits die zu große Bäume erschaffen. Selbst wenn die Entropy dadurch sehr gut wird, ist eine Aufteilung von jedem $n \in N$ in ein einzelnen *node* unbrauchbar. Der Information Gain Ratio würde das erkennen.
- Gini berechnet die Verunreinigung. Ist für uns nur bedingt wichtig.

Ein Decision Tree kann (und sollte) bereinigt werden (*pruning*). Dies passiert durch:

Pre-pruning Vorzeitiges Stoppen bei einem Threshold

Post-pruning Wenn in den Child-nodes summiert mehr misclassifications gemacht werden, als zusammen im Parent, lohnen sie sich nicht.

Continuous Werte können mit $< 500, \geq 500$ in ein Decision Tree eingebaut werden.

ID3 Algorithmus

Sehr simple. Versuche den Tree möglichst klein zu halten, also nehme immer ein Leaf Node wenn du nur noch gleiche target labels, keine weiteren features oder einen pre-pruning threshold überschritten hast. Sonst splitte immer nach dem feature mit dem **höchsten** information gain (nicht doppelt splitten mit dem gleichen feature).

3.2 Clustering

Clustering fällt in die Kategorie des unsupervised learning.

Für clustering müssen wir irgendwie messen, wie weit wir entfernt sind (hier in 2 Dimensionen):

Euclidean Distance $d(x,y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

Manhattan $d(x,y) = |x_1 - y_1| + |x_2 - y_2|$

Chebychev $d(x,y) = \max_i(|x_i - y_i|)$

Minkowski (L^p) $d(x,y) = \sqrt[p]{\sum_i |x_i - y_i|^p}$

k-means

Erzeuge $k \in \mathbb{N}$ *centroid* (zufällig). Weise dann jedem Datenpunkt den nächsten *centroid* zu. Nehme den Mean (Durchschnitt) aus allen zu einem *centroid* gehörigen Datenpunkte und setzte den *centroid* dahin. Weise erneut jedem ...

Es ist garantiert, dass dies irgendwann konvergiert, allerdings hängt das Ergebnis von den Startpositionen der Centroids ab. Ein Problem ist, dass nur „kreisförmige“ Cluster gefunden werden können, insbesondere keine komplizierten verschachtelten Formen.

Eine häufige Error Funktion ist Sum-of-Squares: $E(x, C) = \sum_{i=1}^{k} \sum_{x_j \in C_i} d(x_j, c_i)^2$.

k-medoids

Anstatt eigene *centroids* zu erzeugen nutze manche Datenpunkte (*medoids*).

Weise jedem Datenpunkt einen *medoid* zu. Falls ein Punkt x_i den Error verringert, tausche ihn mit einem *medoid*.

Die Intuition ist richtig und k-medoids ist komplexer (zeitlich) zu berechnen, aber es ist weniger sensitiv zu outliers, da nicht ein einzelner Punkt den *centroid* außerhalb vom wirklichen Cluster ziehen kann. In beiden Fällen ist k natürlich ein Hyperparameter.

3.3 Agglomerative/Dendrogramm

Hier wird die Bottom-Up Technik verwendet um zu clustern.

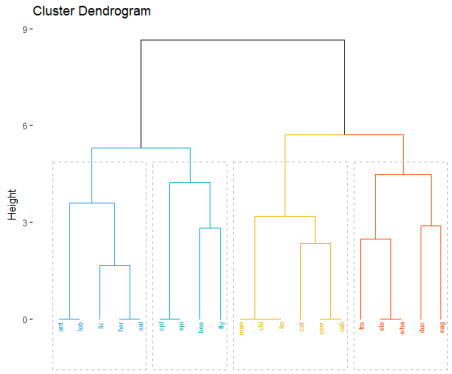

Abbildung 6: Beispiel eines Dendrogramms

1. Erstelle cluster $\mathcal{C}_i$ für jedes x_i
2. Berechne zu allen $d(\mathcal{C}_i, \mathcal{C}_j)$
3. Merge min $d(\mathcal{C}_i, \mathcal{C}_j)$.
4. Schritt 2 bis es nur noch einen Cluster gibt.

Bild Abbildung 6 zeigt die Visualisierung anhand eines Dendrogramm. Ich denke keine Erklärung notwendig.

3.4 DB-Scan

Keine sorge wir sind nicht wieder bei DBIS gelandet.

Zwei Punkte x_i, x_j sind density-connected falls es ein x_k gibt, das mit beiden Verbunden ist. Hieraus werden dann *core-points* gebildet die mit *MinPts* Punkten in ε Nähe sind.

Core points werden dann zu clustern oder clusters werden um core-points erweitert.

Am Ende lassen sich beliebige Formen clustern, anders als k-means (wird aber häufig wegen Einfachheit genutzt).

3.5 Frequent Itemsets

Notation:

- $I = \{I_1, ..., I_D\}$ sind alle möglichen Items
- $A \subseteq I$ ist dann ein *itemset* (*transaction* bei nichtleerem A).
- Ein Dataset X ist ein *multiset* von Transaktionen (mehrere gleiche sind möglich).

Wichtige Metriken sind $\text{support}(A) = \frac{\text{support\_count}(A)}{|X|} = \frac{|[T \in X \mid A \subseteq T]|}{|X|}$

Hierzu ein Beispiel aus der Vorlesung:

$$X = [\{A, B, E\}, \{C, B\}, \{A, D\}, \{A, D, B\}]$$
$$T = \{A, B\} \subseteq I$$

support_count$(A) = |[T_1, T_2]| = 2$, da wir nur $\{A, B\}$ betrachten und nicht nur A, B. Somit ist A, D nicht im support count mit drin.
support$(A) = \frac{2}{4} = \frac{1}{2}$

Wir sagen nun, A ist ein frequent itemset, falls support$(A) \geq$ min_sup. Weil es natürlich viel zu viele Kombinationen gibt, betrachten wir nur die Itemsets, für die es keine gleichen Supersets gibt. Also A ist *closed*, falls support$(A) >$ support(B) für alle $B \supset A$.

Zwei Algorithmen

3.6 Apriori

1. Candidate Generation. Nutze L_k (Frequent Itemsets der Länge k) um die Kandidaten C_{k+1} zu generieren.
2. Pruning. Supersets von *infrequent* Itemsets können nicht *frequent* sein.
3. Testen.

Was macht ihr aber in der Klausur? Ein Beispiel mit min_support $= 2$.

TID	Fruits
1	{Grapes, Apple, Pineapple}
2	{Orange, Apple, Banana}
3	{Grapes, Orange, Apple, Banana}
4	{Orange, Banana}
5	{Grapes, Apple, Banana}

Itemsets	Counts
{Grapes}	3
{Orange}	2
{Apple}	4
{Pineapple}	1
{Banana}	4

Itemsets	Counts
{Grapes}	3
{Orange}	2
{Apple}	4
{Banana}	4

Itemsets
{Grapes, Orange}
{Grapes, Apple}
{Grapes, Banana}
{Orange, Apple}
{Orange, Banana}
{Apple, Banana}

Itemsets	Counts
{Grapes, Orange}	1
{Grapes, Apple}	3
{Grapes, Banana}	2
{Orange, Apple}	2
{Orange, Banana}	3
{Apple, Banana}	3

Itemsets	Counts
{Grapes, Apple}	3
{Grapes, Banana}	2
{Orange, Apple}	2
{Orange, Banana}	3
{Apple, Banana}	3

Dies ist jeweils eine Iteration. Erst selection, dann pruning dann testing.

3.7 FP-Growth

1. Gehe alle Items $I_1, ..., I_D$ durch und sortiere nach Häufigkeit.
2. Lösche alle *non-frequent* items.
3. Gehe durch alle Transaktionen und sortiere die Items nach der obigen Häufigkeit.
4. Baue daraus einen FP-Tree.
    1. Starte bei der Gesamtzahl als Wurzel
    2. Gehe die Transaktionen durch und füge sie an den Baum. Notiere die Häufigkeit der Items.

3. Mine den FP-Tree um die Frequent Items zu bekommen. Hier gibt es Taktik, aber ich würde es raten in der Klausur einfach zu „machen“.

Der FP-Tree kann groß werden, aber falls er ins memory passt, sind nur zwei Durchläufe des Datensatzes nötig.

Association-Rules
Association Rule $A \Rightarrow B$ mit $A \subseteq I, B \subseteq I, A \cap B = \emptyset$.

$$\text{support}(A \Rightarrow B) = \frac{\text{support\_count}(A \cup B)}{\text{support\_count}(\emptyset)}$$

$$\text{conf}(A \Rightarrow B) = \frac{\text{support\_count}(A \cup B)}{\text{support\_count}(A)}$$

conf ist die Confidence.

Da es noch mehr dieser Rules als frequent Itemsets gibt, können wir auch hier wieder pruning und redundant rules löschen.

> A trend appears in several different groups of data but disappears or reverses when these groups are combined

Das ist Simpons-Paradox.

3.8 Sequence Mining

Hierbei gibt es *temporal data*, heißt jede Aktion hat einen Timestamp, eine Case-ID und eine Aktion (z.B. Nutzer 1 registriert sich t_1, Nutzer 2 meldet sich an t_2, etc...).

Hieraus entstehen cases, case $1 = \langle \text{Nutzer registriert}, ... \rangle$.

Wir brauchen erneut eine Relation, das *containment* $A \sqsubseteq B$. Wir gucken einfach ob wir die Events von A in B in der **gleichen Reihenfolge**, aber gegebenenfalls mit Lücken, in B finden.

Der Support ist dann wieder gleich definiert: $\text{support}(P) = \frac{|[S \in X \mid P \sqsubseteq S]|}{|X|}$.

Apriori-All Algorithmus
Wir brauchen alle *litemsets* (doofes wort) also alle $\mathcal{L} = \{A \in I \mid \text{support}(\langle A \rangle) \geq \text{min\_sup}\}$

Nun mappen wir *Sequences* auf die darunter liegenden *litemsets* um. Beispiel $\mathcal{L} = \{\{a\}, \{b\}, \{c\}, \{a, b\}\}$ und die *sequence* $\langle \{a, c\}, \{a, b, c\} \rangle$ wird zu $\langle \{\{a\}, \{c\}\}, \{\{a\}, \{b\}, \{c\}, \{a, b\}\} \rangle$

Nun erstellen wir wieder die Kandidaten und prunen das Ergebnis.

Dann müssen wir noch testen ob min_sup eingehalten wurde.

Wirklich interessante Daten müssen dann mit constraints gefunden werden.

3.9 Time-Series Forecasting

Die Analyse betrachte ich als nicht wirklich relevant und überspringe sie. Falls jemand anders hier eine kleine Zusammenfassung schreibt wäre das vielleicht gut. Nun aber zum *forecasting*

Moving Average Regression der vorherigen aufsummierten Fehler.

3.10 Process-Mining

In Process-Mining geht es erneut um ID-Activity-Timestamp Daten. Wir werden die Event-Daten in 4 Diagrammen modellieren:

Directly-Follows Graph (DFG)

Ein DFG ist wirklich trivial. Ein Beispiel in Abbildung Abbildung 7.

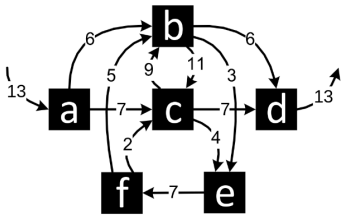

Abbildung 7: Beispiel eines DFG

Und ja wir verknüpfen bereits dagewese Knoten miteinander.

Petri-Nets

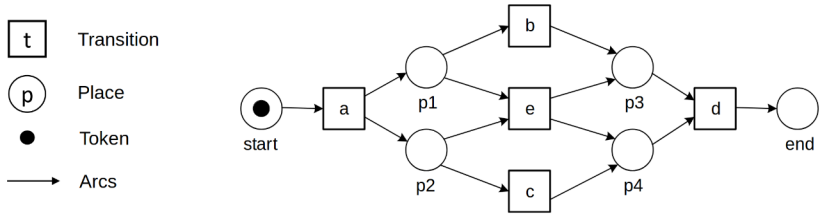

Abbildung 8: Beispiel eines Petri-Nets

Ein Petri-Net besteht aus einem Start, einem Endknoten und mehreren Transitions und Places. Wir fangen beim Start an und legen einen Token hin. Nun gilt die simple Firing Regel: Wenn ein Transition alle Inputs erfüllt hat (dort ein Token liegt), dann tut es auf alle Ausgaben ein Token (und löscht die Eingaben).

Soweit so simple.

Process-Trees

Im Endeffekt bauen wir Process-Trees aus 4 Komponenten zusammen, die sich jeweils in ein Petri-Net übertragen lassen und wodurch sich jedes Petri-Net in ein Process-Tree umwandeln lässt.

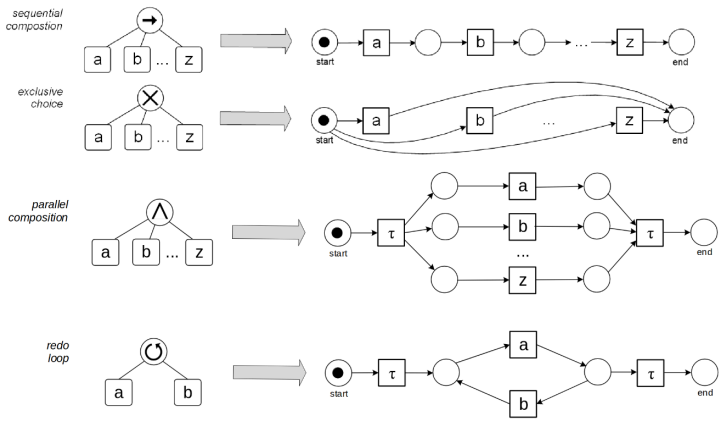

Abbildung 9: Definitionskomponenten des Process-Trees

τ ist hier ein „silent skip“.

Inductive Mining
Wir wollen nun von einem DFG zu einem Process-Tree. Hierfür müssen wir die Komponenten erkennen. Wir betrachten immer zuerst den ganzen DFG und arbeiten uns dann ins Detail. Die Reihenfolge ist hierbei wichtig:
1. *exclusive or*
2. *sequence cut*
3. *parallel cut*
4. *redo-loop cut*

3.11 Replay

Wir müssen wie immer auch bei Petri-Nets die Performance messen. Hierfür haben wir den *fitness* score:

$$\text{fitness}(\sigma, N) = \frac{1}{2}\left(1 - \frac{m}{c}\right) + \frac{1}{2}\left(1 - \frac{r}{p}\right)$$

σ Der Trace der betrachtet wird (eine Sequence)
N Das Netzwerk
p Summe produzierte Tokens
c Summe konsumierte Token
m Missing Tokens immer wenn ein Token für eine Aktivierung fehlt (wir spielen ja ein Trace nach), erschaffen wir eins, aber merken uns die Summe.
r Remaining Tokens. Wenn tokens am Enden übrig bleiben.

Dies können wir für einen gesamten Log ebenso machen. Wir merken uns immer die Summe.

Die Nachteile von solchen Token-based Approaches sind:
- Sie brauchen gelabelte Transitions
- Können misleading Resultate erzeugen.

3.12 Text Mining

Die Steps bei Text-Mining sind:
- Wir haben eine Datenbank
- Daraus extrahieren wir einige Pieces (das können Wörter, Sätze, etc. sein), zusammen ist das ein Corpus
- Den preprocessen wir
- Transformieren ihn zu umgänglicheren Strukturen
- Modellieren ihn
- Und finden somit Patterns

Wir versuchen also aus unstrukturiertem Text strukturierte Daten zu machen.

Hier wird preprocessing wirklich wichtig.

Preprocessing
Wir verwenden Tokenization, Stop-Word removal und Token normalization.

Wir splitten Sätze in Tokens (z.B. anhand von Lehrzeichen) müssen dabei aber auf einige Dinge achten: „He's“ sollte z.B. zu „He“, „is“ werden oder vielleicht „New York City.“ -> „New York City“ statt „New“, „York“, „City.“

Dann entfernen wir unnötige Informationen (Stop-Words) wie: Artikel, Präpositionen, etc. Die Informationen könnten natürlich wichtig sein, ist immer Abwägungssache.

Schließlich wollen wir noch die Tokens auf einen Nenner bringen. Hierfür wenden wir an:

Stemming Wörter nur mit deren Wortstamm. „verzweifeln -> verzweifel“ (das muss kein Wort sein)

Lemmatization Dasselbe wie Stemming, nur das wir ein „base“ Wort aus dem Wörterbuch benutzen z.B. „verzweifelt“

Modellierung

Hierfür gibt es verschiedene Models:

Bag of Words *multiset* der Wörter in einem Corpus. Wir verlieren somit die Reihenfolge

Document Term Matrix Wir listen alle Wörter in columns auf und haben als Reihen die Dokumente, die Zellen sind einfach die Anzahl.

Term Frequency Anzahl von w in Dokument d
$\mathrm{tf}(w, d) = |d_w|$

Inverse Document Frequency Je unwahrscheinlicher das Wort, desto höher der Wert. Stellt besonders informationsbringende Wörter heraus.
$\mathrm{idf}(w, c) = \log_2\left(\frac{|c|}{\text{Anzahl der Dokumente in } c \text{ die } w \text{ enthalten}}\right)$

TF-IDF Score Kombination aus den beiden $\mathrm{tfidf}(w, d, c) = \mathrm{tf}(w, d) \cdot \mathrm{idf}(w, c)$

Sonstiges

Die N-Gram Anwendungen sind Wissenswert. Nutze die letzten N Wörter, um das nächste vorherzusagen. Das rechnen wir mit Wahrscheinlichkeiten aus, aber da wir nicht jede Wortkombination in unseren Corpora haben, müssen wir die Wahrscheinlichkeitsverteilung smoothen, um nicht nur Nullen zu haben.

Häufig werden auch statt Wörtern ein Vektor encoding genutzt, das von einem Neural Network produziert wird. Das ist einfach effizienter.

3.13 Sonstiges Data Science Zeugs

Preprocessing ist sehr häufig sehr wichtig. Vor allem bei Big-Data müssen die Datenmengen reduziert werden:

Feature Reduction Autoencoders sind NNs die eine automatische effizientere encoding der Daten machen. Principal Component Analysis kombiniert die features zusammen.
Aber auch Feature Selection, also aussortieren unnötiger Features, gehört dazu.

Instance Reduction Sampling (z.B. nehme N zufällige instances)

4 Evaluation and AutoML/DS

Wir wollen beantworten:

- Wie gut ist das ML Model?
- Wie gut könnte es sein?

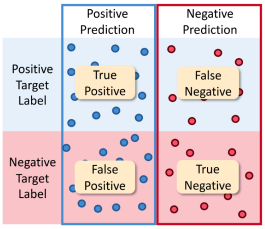

Abbildung 10: Beispiel Confusion Matrix

Wie bei Statistischen Tests und Studien gibt es hier ein Falsch positives FP, Falsch negatives FN und Wahr positives TP und Wahr negatives TN Ergebnis. Abbildung Abbildung 10 zeigt das im binomial Fall.

In einem Multinomial Fall, muss man sich ein Feature jeweils raus suchen und sich daraus eine Binomial Matrix bilden. So wie es Sinn ergibt.

Darauf aufbauend haben wir erneut einige Scores:
Accuracy $\frac{\mathrm{TP}+\mathrm{TN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$
Misclassification Rate $\frac{\mathrm{FP}+\mathrm{FN}}{\mathrm{TP}+\mathrm{TN}+\mathrm{FP}+\mathrm{FN}}$
True Positiv Rate $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$
False Negative Rate $\frac{\mathrm{FN}}{\mathrm{TP}+\mathrm{FN}}$
True Negative Rate $\frac{\mathrm{TN}}{\mathrm{TN}+\mathrm{FP}}$
False Positive Rate $\frac{\mathrm{FP}}{\mathrm{TN}+\mathrm{FP}}$
Recall $= \mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$
Precision $\frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}}$
F_1 $2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recal}}$
Wie viele Trainingsdaten brauchen wir und wie sind die aufgeteilt?

Wir benötigen ein Trainingsset soweit ist's klar. Aber um overfitting während des Trainings zu reduzieren, benötigen wir ein Validation set. Hiermit können wir abbrechen bei overfitting, oder Hyperparameter Optimierungen ausführen.

Aber schließlich brauchen wir noch ein Testing set um die Evaluation des Modells auszuführen. Das Verhältnis ist variabel, aber (50%, 20%, 30%) oder (40%,20%,40%) wurde in der Vorlesung erwähnt. Diese Validierung machen wir meistens mit verschiedenen Modellen gleichzeitig und wählst dann mit den folgenden Methoden die Testdaten aus:
k-Fold Cross Validation Wähle für $\frac{N}{k}$ große Daten aus. N ist die gesamt Testdaten Größe und wir haben k unterschiedliche Folds.
Jackknifing Wie k-Fold aber wähle nur eine Testdaten *instance*.
Bootstrapping Wähle $m \in \mathbb{N}$ *instances* zufällig aus.

Hier kann ebenfalls auf Stochastische Tests (t-tests) zurückgegriffen werden, um die Signifikanz von Abweichungen zu überprüfen.

Da FP und FN unterschiedliche Kosten haben können wenden wir eine Profit-Matrix an, die jeweils die unterschiedlichen Kategorien gewichtet.

Für die zweite Frage gucken wir uns die ROC-Kurve an.

Hier plotten wir die TPR auf der y und die FPR auf der X Achse. Unter der Diagonale ist alles schlechter als random guessing, also kann man hier die Entscheidungen umdrehen und erhält ein gutes Model. Sonst ist ein Kurve die Stark zur 1 auf der X Achse wandert und dann da bleibt besser.

4.1 AutoML

Hyperparameter Optimization

Es gibt bei uns grob 4 Methoden:
- Random Search
- Grid Search
- Bayesian Optimization
- Multi Fidelity Bandit