Deep Learning Lecture Notes

Zusammenfassung der Vorlesungseinheiten

Stand: Heute

Vorlesungsskript: Lineare Regression & MSE

1. Einführung

Die Lineare Regression ist der "Hello World"-Algorithmus des Supervised Learning. Ziel ist es, einen kontinuierlichen Zielwert (Target y) basierend auf einem Eingabewert (Feature x) vorherzusagen.

Beispiele:

- Wohnfläche (x) $\rightarrow$ Hauspreis (y)
- Lernzeit (x) $\rightarrow$ Klausurnote (y)

2. Das Modell

Wir nehmen an, dass der Zusammenhang linear ist. Unsere Hypothese $h(x)$ lautet:

$$ \hat{y} = w \cdot x + b $$

- w (Weight/Slope): Wie stark ändert sich y, wenn x steigt?
- b (Bias/Intercept): Welchen Wert hat y, wenn $x=0$ ist?
- $\hat{y}$: Die Vorhersage unseres Modells (im Gegensatz zum echten y).

3. Die Kostenfunktion (Loss Function)

Um die besten Werte für w und b zu finden, müssen wir definieren, was "gut" bedeutet. In der Regression nutzen wir meist den **Mean Squared Error (MSE)**.

Warum quadrieren?

Der Fehler für einen einzelnen Punkt ist $e_i = y_i - \hat{y}_i$.

1. **Vorzeichen eliminieren**: Würden wir nur summieren ($\sum (y - \hat{y})$), könnten sich Fehler von +10 und -10 zu 0 aufheben. Das Modell wäre "perfekt", obwohl es falsch liegt.
2. **Bestrafung großer Fehler**: Durch das Quadrat $(10)^2 = 100$ vs $(1)^2 = 1$ fallen Ausreißer stärker ins Gewicht.

Die Formel

$$ J(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (w x_i + b))^2 $$

4. Optimierung (Wie lernt die Maschine?)

Wir suchen das Paar (w, b), das $J(w, b)$ minimiert.

Analytische Lösung (Normal Equation)

Für einfache Lineare Regression gibt es eine geschlossene Formel: $$ w = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} $$ Das ist schnell für kleine Datenmengen, wird aber bei Millionen von Features (Matrix-Inversion) zu langsam.

Numerische Lösung (Gradient Descent)

Das ist der Standard für Deep Learning.

1. Starte mit zufälligem w, b.
2. Berechne den Gradienten (die Steigung der Fehlerlandschaft).
3. Gehe einen kleinen Schritt entgegen dem Gradienten.
4. Wiederhole.

5. Praxis-Checkliste

Frage	Antwort
Wann nutzen?	Bei einfachen Zusammenhängen, als Baseline-Modell.
Vorteile	Sehr schnell, leicht interpretierbar (w zeigt Wichtigkeit).
Nachteile	Kann nur lineare Zusammenhänge lernen (keine Kurven ohne Feature Engineering).
Annahmen	Homoskedastizität (gleiche Varianz der Fehler), keine Multikollinearität.

6. Übungsaufgabe

Öffnen Sie `code/lab.py` und versuchen Sie:

1. Das Rauschen (`noise`) in der Datengenerierung zu erhöhen. Wie verändert sich der MSE?
2. Implementieren Sie eine Funktion `calculate_mae` (Mean Absolute Error) und vergleichen Sie.

Vorlesungsskript: Logistische Regression

1. Motivation: Warum nicht einfach eine Gerade?

Angenommen, wir wollen vorhersagen, ob ein Kunde kauft ($y=1$) oder nicht ($y=0$). Wenn wir Lineare Regression ($y = wx+b$) nutzen, kann das Modell Werte wie 1.5 oder -0.3 ausgeben.

- Was bedeutet eine Kaufwahrscheinlichkeit von -30%?
- Was bedeutet 150%?

Wir brauchen eine Funktion, die beliebige Eingaben auf das Intervall $[0, 1]$ "quetscht".

2. Die Sigmoid-Funktion

Die Lösung ist die logistische Funktion (Sigmoid):

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

- Wenn z sehr groß ist ($z \to \infty$), wird $e^{-z} \to 0$, also $\sigma(z) \to 1$.
- Wenn z sehr klein ist ($z \to -\infty$), wird $e^{-z} \to \infty$, der Nenner riesig, also $\sigma(z) \to 0$.
- Wenn $z = 0$, ist $e^0 = 1$, also $\sigma(0) = 0.5$.

Unser Modell wird also: $$ \hat{y} = \sigma(w \cdot x + b) $$

3. Interpretation

Wir interpretieren $\hat{y}$ als die bedingte Wahrscheinlichkeit, dass die Klasse 1 ist: $$ \hat{y} = P(y=1 | x) $$

Die Entscheidung (Klassifikation) treffen wir meist bei 0.5:

- Wenn $\hat{y} \ge 0.5 \rightarrow$ Klasse 1
- Wenn $\hat{y} < 0.5 \rightarrow$ Klasse 0

4. Die Kostenfunktion: Log Loss (Cross-Entropy)

Bei der Linearen Regression nutzten wir MSE (quadratischer Fehler). Würden wir das hier tun, bekämen wir eine "wellige" Fehlerlandschaft (nicht-konvex), in der der Gradient Descent stecken bleiben kann.

Stattdessen nutzen wir die **Binary Cross-Entropy**:

$$ J(w,b) = - \frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1-y_i) \log(1-\hat{y}_i)] $$

Intuition

Betrachten wir einen einzelnen Fall:

- Wenn das wahre Label $y=1$ ist, fällt der zweite Term weg. Wir wollen $\log(\hat{y})$ maximieren (also $\hat{y}$ nahe 1).

- Wenn das wahre Label $y=0$ ist, fällt der erste Term weg. Wir wollen $\log(1-\hat{y})$ maximieren (also $\hat{y}$ nahe 0).
- $\log(0)$ geht gegen $-\infty$. Das bedeutet: Wenn das Modell sich "sicher" ist (z.B. $\hat{y}=0.99$), aber falsch liegt ($y=0$), ist die Strafe (Loss) extrem hoch.

5. Praxis-Checkliste

Eigenschaft	Logistische Regression
Typ	Klassifikation (binär, erweiterbar auf Multiclass via Softmax).
Linearität	Die Entscheidungsgrenze (Decision Boundary) ist linear!
Output	Kalibrierte Wahrscheinlichkeiten.
Wichtig	Features sollten skaliert sein (StandardScaler), da w direkt in den Exponenten eingeht.

6. Übungsaufgabe (`code/lab.py`)

1. Implementieren Sie die `sigmoid` Funktion manuell.
2. Berechnen Sie den Log Loss für folgende Vorhersage:
    - Wahrheit: [1, 0, 1]
    - Modell A: [0.9, 0.1, 0.8] (Gut)
    - Modell B: [0.6, 0.4, 0.6] (Unsicher)
    - Modell C: [0.1, 0.9, 0.1] (Falsch und sicher -> Katastrophe)

Vorlesungsskript: Softmax & Multiclass Classification

1. Motivation

Die Logistische Regression (Unit 02) nutzt die Sigmoid-Funktion, um einen Wert zwischen 0 und 1 zu erzeugen ($P(y=1)$). Das funktioniert super für "Spam vs. Ham".

Aber was ist mit der Ziffernerkennung (0-9) oder der Klassifikation von Blumenarten (Iris Setosa, Versicolor, Virginica)? Wir brauchen ein Modell, das K verschiedene Wahrscheinlichkeiten ausgibt, die sich zu 100% (1.0) summieren.

2. Das Modell (Logits)

Statt einem Gewichtsvektor w haben wir nun eine Gewichtsmatrix W. Man kann sich das so vorstellen, dass wir für jede der K Klassen eine eigene lineare Funktion haben.

Für jede Klasse k berechnen wir einen Score (genannt **Logit**): $$ z_k = w_k \cdot x + b_k $$

Diese Scores können beliebige Werte annehmen (z.B. -5.0, 12.4, 0.0).

3. Die Softmax-Funktion

Um aus den Scores (z) interpretierbare Wahrscheinlichkeiten ($\hat{y}$) zu machen, nutzen wir die Softmax-Funktion. Sie verstärkt die Unterschiede (große Werte werden noch größer) und normiert alles.

$$ \hat{y}_i = \text{Softmax}(z)*i = |*frac{e^{z_i}}{|*sum{j=1}^K e^{z_j}} $$

Beispiel: Scores $z = [2.0, 1.0, 0.1]$

1. Exponenzieren (e^z): $[7.39, 2.71, 1.10]$
2. Summieren: $7.39 + 2.71 + 1.10 = 11.2$
3. Normieren (Teilen durch Summe): $[0.66, 0.24, 0.10]$

Ergebnis: Klasse 1 hat 66% Wahrscheinlichkeit.

4. Loss Funktion: Categorical Cross-Entropy

Wie trainieren wir das? Wir vergleichen den vorhergesagten Wahrscheinlichkeitsvektor $\hat{y}$ mit dem wahren Vektor y (One-Hot Encoded).

Wenn das Bild eine "Klasse 1" (Index 0) ist, sieht y so aus: $[1, 0, 0]$. Die Formel für den Loss ist:

$$ J = - \sum_{k=1}^K y_k \log(\hat{y}_k) $$

Da im One-Hot-Vektor y überall Nullen stehen außer bei der richtigen Klasse t, fällt die Summe weg und es bleibt nur:

$$ J = - \log(\hat{y}_t) $$

Das heißt: Wir schauen uns nur die Wahrscheinlichkeit an, die das Modell für die **richtige** Klasse vorhergesagt hat, und nehmen davon den negativen Logarithmus.

- Ist $\hat{y}_t \approx 1$ (sicher richtig), ist $-\log(1) = 0$. (Kein Fehler)
- Ist $\hat{y}_t \approx 0$ (falsch), ist $-\log(0) \to \infty$. (Großer Fehler)

5. Numerische Stabilität (Log-Sum-Exp)

Ein technisches Detail für die Implementierung: e^{1000} führt im Computer zu einem Überlauf (Infinity/NaN).

Der Trick: Mathematisch gilt: $\frac{e^{z_i}}{ \sum e^{z_j} } = \frac{e^{z_i - C}}{ \sum e^{z_j - C} }$ für eine Konstante C. Wir wählen $C = \max(z)$. Damit ist der größte Exponent im Vektor genau 0 ($e^0=1$), und alle anderen sind negativ (e^{-x} ist klein, aber stabil).

6. Übungsaufgabe (`code/lab.py`)

1. Implementieren Sie die `softmax` Funktion.
2. Berechnen Sie manuell den Loss für ein Beispiel.
3. Beobachten Sie, wie sich der Loss ändert, wenn das Modell "unsicherer" wird.

Vorlesungsskript: Optimierung

1. Die Loss-Landschaft

Das Training eines Modells ist im Grunde die Suche nach dem tiefsten Punkt in einer riesigen Gebirgslandschaft.

- **Koordinaten**: Die Gewichte w des Modells.
- **Höhe**: Der Loss $J(w)$.
- **Ziel**: Finde w, wo $J(w)$ minimal ist.

2. Gradient Descent (GD)

Da wir die Landschaft nicht komplett sehen (zu viele Dimensionen), tasten wir uns vor. Der **Gradient** $\nabla J(w)$ zeigt immer in die Richtung des steilsten Anstiegs. Wir gehen also in die entgegengesetzte Richtung.

$$ w_{neu} = w_{alt} - \eta \cdot \nabla J(w_{alt}) $$

- η (Eta): Die **Learning Rate**. Einer der wichtigsten Hyperparameter.

3. Stochastic Gradient Descent (SGD)

Berechnet man den Gradienten über *alle* Daten (Batch GD), ist das sehr präzise, aber extrem langsam. **SGD** nimmt nur ein einziges Beispiel (oder eine kleine Mini-Batch) pro Schritt.

- Vorteil: Viel schneller, weniger Speicher.
- Nachteil: Der Weg ist sehr "zittrig" (stochastisch). Das Rauschen kann aber helfen, aus flachen lokalen Minima zu entkommen.

4. Momentum

In engen Tälern (Ravines) springt SGD oft wild hin und her, ohne vorwärts zu kommen. **Momentum** löst das, indem es eine "Geschwindigkeit" v einführt.

$$ v_{neu} = \gamma \cdot v_{alt} + \eta \cdot \nabla J(w) $$ $$ w_{neu} = w_{alt} - v_{neu} $$

- γ (Gamma): Reibung (meist 0.9).
- Effekt: Wenn der Gradient immer in die gleiche Richtung zeigt, werden wir schneller. Bei Zick-Zack-Bewegungen mitteln sich die Querschläger raus.

5. Adam (Adaptive Moment Estimation)

Adam kombiniert die besten Ideen:

1. **Momentum**: (Wie oben) für die Richtung.
2. **RMSProp**: Skaliert die Lernrate basierend auf der Varianz der Gradienten.

Das bedeutet: Parameter, die selten Updates bekommen, erhalten größere Schritte. Parameter, die stark schwanken, werden gebremst. Adam ist heute der "Default"-Optimierer für fast alle Deep Learning Aufgaben.

6. Übungsaufgabe (`code/lab.py`)

Wir simulieren eine einfache 2D-Funktion $f(x, y) = x^2 + 10y^2$. Das ist ein enges Tal.

- Beobachten Sie, wie "Vanilla GD" hin und her springt.
- Beobachten Sie, wie "Momentum" direkt zur Mitte steuert.

Vorlesungsskript: CNN Basics

1. Warum nicht einfach Dense Layers?

Ein Bild mit 1000×1000 Pixeln hat 1 Million Eingabewerte. Verbinden wir das mit einer Hidden Layer von 1000 Neuronen, bräuchten wir $10^6 \times 10^3 = 1$ Milliarde Gewichte. Das ist nicht trainierbar (und speicherintensiv).

Zudem: Ein Dense Layer weiß nicht, dass Pixel (0,0) und Pixel (0,1) Nachbarn sind. Für ihn sind das einfach x_0 und x_1 ohne räumlichen Bezug.

2. Die Convolution (Faltung)

Die Kernidee: Wir suchen nach lokalen Mustern. Ein **Kernel** (oder Filter) ist eine kleine Matrix (z.B. 3×3). Wir schieben diesen Kernel Schritt für Schritt über das Bild.

$$ (I * K)*{x,y} = |sum*{i=0}^{2} \sum_{j=0}^{2} I_{x+i, y+j} \cdot K_{i,j} $$

- Das ist im Grunde ein Skalarprodukt zwischen dem kleinen Bildausschnitt und dem Filter.
- Ist das Muster im Bild ähnlich zum Filter, ist das Ergebnis groß (hohe Aktivierung).

3. Filter lernen

In der klassischen Bildverarbeitung (Photoshop) hat man Filter fest programmiert (z.B. Sobel-Filter für Kanten). In CNNs sind die Werte im Kernel **lernbare Gewichte**. Das Netz lernt selbst, welche Filter nützlich sind (Kanten, Kreise, Texturen).

4. Wichtige Parameter

- **Kernel Size**: Meist 3×3 oder 5×5.
- **Stride**: Die Schrittweite. Stride 1 = Pixel für Pixel. Stride 2 = Wir überspringen jeden zweiten (Bild wird kleiner).
- **Padding**: Rahmen aus Nullen um das Bild, damit es nicht kleiner wird ("Same Padding").

5. Pooling

Nach der Convolution folgt oft eine Pooling-Schicht. Ziel: Datenmenge reduzieren und Abstraktion erhöhen.

Max Pooling (2×2): Wir schauen uns 2×2 Blöcke an und behalten nur den **größten** Wert.

- Wirft 75% der Daten weg.
- Behält nur die Info "Hier war irgendwo eine starke Kante", egal wo genau im 2x2 Block.

6. Architektur eines CNN

Typischer Aufbau (wie Lego):

1. Input Image
2. Conv Layer (findet Features) + ReLU (Aktivierung)

3. Pool Layer (macht kleiner)
4. Conv Layer (findet komplexere Features aus den vorherigen) + ReLU
5. Pool Layer
6. Flatten (alles in einen Vektor)
7. Dense Layer (Klassifikation)
8. Output (Softmax)

Vorlesungsskript: Attention & Transformer

1. Rückblick: Sequenzverarbeitung vor 2017

Früher nutzte man RNNs (Recurrent Neural Networks). Diese verarbeiten Wort für Wort sequenziell.

- **Problem 1 (Bottleneck):** Der gesamte Inhalt des Satzes muss in einen einzigen Vektor ("Hidden State") gepresst werden. Bei langen Sätzen geht Information verloren.
- **Problem 2 (Speed):** Man kann Wort 10 erst berechnen, wenn Wort 9 fertig ist. Keine Parallelisierung auf GPUs möglich.

2. Die Idee von Attention

Statt nur den letzten Zustand zu nutzen, erlauben wir dem Modell, auf **alle** vorherigen Zustände zuzugreifen. Es berechnet für jeden Output eine Gewichtung: "Wie wichtig ist Input-Wort j für Output-Wort i?"

3. Self-Attention (Der Kern des Transformers)

Self-Attention wendet dieses Prinzip auf den Satz selbst an. Jedes Wort schaut auf jedes andere Wort im gleichen Satz, um seinen Kontext zu verstehen.

Die Metapher: Datenbank-Suche

Für jedes Wort erstellen wir drei Vektoren:

1. **Query (Q):** Wonach suche ich? (z.B. "Ich bin ein Pronomen, worauf beziehe ich mich?")
2. **Key (K):** Was biete ich an? (z.B. "Ich bin ein Substantiv, männlich.")
3. **Value (V):** Was ist mein Inhalt? (Die eigentliche Wortbedeutung).

Die Formel (Scaled Dot-Product Attention)

$$ \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V $$

1. QK^T: Berechnet die Ähnlichkeit (Dot Product) zwischen meiner Suche (Q) und allen Angeboten (K).
2. softmax: Wandelt die Scores in Wahrscheinlichkeiten um (Summe = 1). Das sind die **Attention Weights**.
3. $\cdot V$: Wir berechnen den gewichteten Durchschnitt der Inhalte (V).

4. Multi-Head Attention

Ein einziger Attention-Mechanismus reicht oft nicht. Ein "Head" achtet vielleicht auf Grammatik, ein anderer auf semantische Bezüge. Deshalb nutzen Transformer mehrere Heads parallel und konkatenieren die Ergebnisse.

5. Positional Encoding

Da der Transformer alle Wörter gleichzeitig sieht (keine Sequenz), weiß er nicht, ob "Mann beißt Hund" oder "Hund beißt Mann" gemeint ist. Lösung: Wir addieren zu jedem Wortvektor ein Muster, das die Position codiert (Sinus/Cosinus-Wellen verschiedener Frequenzen).

6. Architektur-Überblick (GPT vs. BERT)

- **Encoder-Only (z.B. BERT):** Sieht den ganzen Satz (links und rechts). Gut für Klassifikation, Sentiment.
- **Decoder-Only (z.B. GPT):** Sieht nur, was bisher geschah (maskierte Attention). Gut für Textgenerierung.
- **Encoder-Decoder (z.B. T5, Original Transformer):** Gut für Übersetzung.

7. Übungsaufgabe (`code/lab.py`)

Wir simulieren Self-Attention mit NumPy:

1. Wir definieren Dummy-Embeddings für 4 Wörter.
2. Wir berechnen die Attention-Scores.
3. Wir visualisieren, welches Wort auf welches andere "achtet".

Vorlesungsskript: Bias-Variance Tradeoff

1. Das Ziel des Lernens

Wir wollen eine Funktion $f(x)$ lernen, die auf **neuen** Daten gut funktioniert (Generalisierung). Der Fehler auf neuen Daten setzt sich aus drei Komponenten zusammen.

2. Die drei Komponenten des Fehlers

A. Bias (Verzerrung)

- **Definition**: Der Fehler, der durch zu starke Vereinfachung entsteht.
- **Beispiel**: Wir versuchen, eine Parabel mit einer geraden Linie zu modellieren. Egal wie viele Daten wir haben, die Gerade wird die Parabel nie perfekt treffen.
- **Symptom**: Hoher Training-Error (Underfitting).

B. Variance (Varianz)

- **Definition**: Der Fehler, der entsteht, weil das Modell zu stark auf die spezifischen Trainingsdaten reagiert.
- **Beispiel**: Wir verbinden alle Punkte mit einem Zick-Zack-Polynom. Wenn wir neue Datenpunkte ziehen würden, sähe das Polynom komplett anders aus.
- **Symptom**: Niedriger Training-Error, aber hoher Test-Error (Overfitting).

C. Irreducible Error (Noise)

- **Definition**: Rauschen in den Daten (Messfehler, fehlende Informationen).
- **Eigenschaft**: Kann nicht durch bessere Modelle entfernt werden. Es ist die untere Schranke für den Fehler.

3. Die Formel

$$ E[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2 $$

- Bias^2: Quadratischer Abstand der durchschnittlichen Vorhersage zur Wahrheit.
- Var: Streuung der Vorhersagen um ihren eigenen Durchschnitt.
- σ^2: Das Rauschen.

4. Der Tradeoff

Wenn wir die Modellkomplexität erhöhen:

1. **Bias sinkt**: Das Modell kann die Wahrheit besser abbilden.
2. **Variance steigt**: Das Modell wird anfälliger für Rauschen.

Wir suchen das Minimum der Summe (U-Kurve).

5. Diagnose mit Lernkurven

Wir plotten den Fehler über der Anzahl der Trainingsbeispiele (m).

- **High Bias (Underfitting)**:
    - Training Error ist hoch.
    - Validation Error ist hoch.
    - Beide Kurven nähern sich an, aber auf hohem Niveau.
    - *Lösung*: Komplexeres Modell, mehr Features.
- **High Variance (Overfitting)**:
    - Training Error ist niedrig.
    - Validation Error ist viel höher.
    - Große Lücke zwischen den Kurven.
    - *Lösung*: Mehr Daten, Regularisierung, einfacheres Modell.

6. Übungsaufgabe (`code/lab.py`)

Wir führen ein Experiment durch:

1. Die "Wahrheit" ist eine Sinus-Kurve.
2. Wir ziehen 100 verschiedene kleine Datensätze aus dieser Wahrheit (mit Rauschen).
3. Wir trainieren jeweils ein einfaches (Grad 1) und ein komplexes Modell (Grad 9).
4. Wir visualisieren: Das einfache Modell ist immer ähnlich falsch (Bias). Das komplexe Modell zappelt wild umher (Variance).

Vorlesungsskript — Ensembling: Random Forests & Gradient Boosting

Dieses Skript begleitet die Vorlesungseinheit "Ensembling". Es vertieft die mathematischen Grundlagen, bietet Intuitionen zu den Algorithmen und fasst Best Practices für den produktiven Einsatz zusammen.

1. Kernaussagen & Motivation

- Ziel: Ensembling kombiniert mehrere Modelle, um Gesamtleistung und Robustheit zu verbessern (Varianz- und/oder Bias-Reduktion).
- Random Forest (RF): Bagging (Bootstrap + Aggregation) von Entscheidungsbäumen → primär Varianzreduktion; robust gegenüber Rauschen und wenig Feature-Engineering.
- Gradient Boosting (GB): sequentielle Fehleranpassung (Additive Modelle) → reduziert Bias, erfordert mehr Hyperparametertuning und Regularisierung.

2. Zentrale Formeln & Intuitionen

- Bagging-Ensemble (Vorhersage): $$\hat{f}(x)=\frac{1}{M}\sum_{m=1}^{M} f_m(x)$$
- Gradient Boosting (Update-Skizze): $$F_{0}(x)=\arg\min_{\gamma}\sum_i L(y_i,\gamma),\ F_{t+1}(x)=F_t(x)+\eta,h_t(x)$$ wobei h_t auf den negativen Gradienten (Residuen) approximiert wird und η die Lernrate ist.
- Feature Importance (RF): mittlere Reduktion der Impurity; Permutation Importance als robustere Alternative.

3. Experiment: Aufbau & Ergebnisse

- Datensatz: `sklearn.datasets.load_iris` (oder `load_breast_cancer` für binäre Klassifikation).
- Vorgehen: Train/Test-Split (70/30), RF und GB mit Standard-Hyperparametern (z.B. 100 Bäume), Accuracy & Classification Report; zusätzlich Cross-Validation (5-fold).
- Wichtige Visualisierungen: Feature importances (Barplot), Lernkurve (Training/Validation Accuracy vs. Trainingsgröße), Vergleichs-Metriken (Accuracy, Precision, Recall, ggf. ROC/AUC für binär).

4. Praktische Hinweise & Empfehlungen

- Hyperparameter (Kurz):
  - RF: `n_estimators`, `max_depth`, `max_features`.
  - GB: `n_estimators`, `learning_rate`, `max_depth`, `subsample`.
- Vermeidung von Overfitting: frühes Stoppen (early_stopping_rounds bei GB), reduzierte `max_depth`, Subsampling.
- Wann welches Modell?: RF als schnelle, robuste Baseline; GB für höhere Performance nach sorgfältigem Tuning.

Weiterführende Links

- scikit-learn: https://scikit-learn.org/stable/modules/ensemble.html
- Friedman, Hastie, Tibshirani — The Elements of Statistical Learning

Praktischer Zusatz: Das beiliegende `code/demo.ipynb` und `code/demo_run.py` führen das kleine Experiment aus und speichern Plots in `assets/`.