Zwischen Interpretation und Berechnung

Hierarchische Grammatikinduktion als Explikation latenter Sequenzstrukturen in Verkaufsgesprächen

Paul Koop

Juni/Juli 1994 & 2024/2026

Zusammenfassung

Die qualitative Sozialforschung steht gegenwärtig vor einem methodologischen Dilemma: Einerseits versprechen generative KI-Systeme eine bislang unerreichte Skalierung interpretativer Arbeitsschritte, andererseits entziehen sie sich durch ihre stochastische Natur der klassischen Validierungslogik qualitativer Forschung. Der vorliegende Beitrag argumentiert, dass dieses Dilemma durch eine Rückbesinnung auf formalisierende Ansätze aufgelöst werden kann. Als konkreten Lösungsansatz entwickelt der Beitrag die **Algorithmisch Rekursive Sequenzanalyse (ARS) in ihrer Version 3.0**, ein Verfahren, das Interpretationsprozesse in eine hierarchische Grammatik überführt und damit nicht nur sequenzielle Übergänge, sondern auch komplexe Interaktionsmuster als interpretative Kategorien expliziert. Die Verbindung zur aktuellen Diskussion um **Explainable AI (XAI)** erweist sich dabei als doppelt fruchtbar: Sie stellt ein begriffliches Instrumentarium bereit, um die Güte qualitativer Interpretationen zu reflektieren, und erinnert daran, dass Erklärbarkeit kein Luxus, sondern eine Notwendigkeit ist – in der Technik wie in der Wissenschaft. Die empirische Anwendung an acht Transkripten von Verkaufsgesprächen demonstriert die Leistungsfähigkeit des Verfahrens zur Bildung interpretativer Kategorien durch hierarchische Kompression.

Inhaltsverzeichnis

1 Einleitung: Das Paradoxon qualitativer Forschung im Zeitalter generativer KI

Die qualitative Sozialforschung steht gegenwärtig vor einem methodologischen Dilemma. Einerseits versprechen generative KI-Systeme eine bislang unerreichte Skalierung interpretativer Arbeitsschritte. Andererseits entziehen sich eben diese Systeme durch ihre stochastische Natur der klassischen Validierungslogik qualitativer Forschung. Wo diese traditionell auf die detaillierte Offenlegung des Codierprozesses und die intersubjektive Nachvollziehbarkeit setzt, tritt nun ein blinder Verlass auf die vermeintliche „Emergenz“ neuronaler Netze.

Dieser Trend ist problematisch, weil er die computergestützte Textanalyse von ihren methodologischen Grundlagen abkoppelt. Zugleich aber verweist er auf ein Defizit, das die qualitative Forschung selbst betrifft: Sie verfügt über kein formalisiertes Vokabular, um ihre Interpretationsprozesse für algorithmische Verfahren anschlussfähig zu machen. Die Folge ist eine Wahl zwischen zwei unbefriedigenden Optionen: entweder Verzicht auf Skalierung oder Preisgabe methodologischer Kontrolle.

Der vorliegende Beitrag argumentiert, dass dieses Dilemma durch eine Rückbesinnung auf formalisierende Ansätze aufgelöst werden kann, die in der Tradition der Textanalyse bereits angelegt waren. Als konkreten Lösungsansatz entwickelt der Beitrag die **Algorithmisch Rekursive Sequenzanalyse (ARS) in ihrer Version 3.0**, ein Verfahren, das Interpretationsprozesse nicht nur in eine sequenzielle Übergangsgrammatik, sondern in eine hierarchische Grammatik mit expliziten Nonterminalen überführt. Diese Nonterminale werden als **interpretative Kategorien** verstanden, die durch wiederholte Sequenzmuster induziert werden – analog zur Bildung neuer Variablen bei Termumformungen, bis nur noch ein Symbol übrig bleibt.

Die Pointe dieses Ansatzes liegt in seiner Verbindung zu aktuellen Diskussionen um **Explainable Artificial Intelligence (XAI)**. XAI hat sich als Antwort auf die Opazität neuronaler Netze entwickelt (Samek & Müller, 2019; Barredo Arrieta et al., 2020). Die zentrale Einsicht lautet: Wer die Entscheidungen komplexer KI-Systeme nicht nachvollziehen kann, kann ihnen nicht vertrauen – und darf sie in sicherheitskritischen Bereichen nicht einsetzen (Weller, 2019). Diese Einsicht, so die These des Beitrags, lässt sich produktiv auf die qualitative Forschung wenden: Auch sie benötigt Verfahren, die ihre Interpretationsprozesse erklärbar machen. Die ARS 3.0 versteht sich als ein solches Verfahren – als Beitrag zu einer **erklärbaren qualitativen Forschung**, die die methodologischen Standards der Disziplin wahrt und zugleich für algorithmische Modellierung öffnet.

Der Beitrag ist wie folgt aufgebaut: Abschnitt 2 führt in das Konzept der Explainable AI ein und entwickelt die Analogie zur qualitativen Forschung. Abschnitt 3 stellt die ARS 3.0 in ihrer methodischen Architektur dar, mit besonderem Fokus auf die hierarchische Grammatikinduktion. Abschnitt 4 dokumentiert die empirische Anwendung an acht Transkripten von Verkaufsgesprächen. Abschnitt 5 reflektiert die Ergebnisse im Lichte der XAI-Kriterien. Abschnitt 6 zieht ein Fazit und zeigt Perspektiven auf.

2 Explainable AI: Begriff, Entwicklung und methodologische Relevanz

2.1 Entstehung und Grundgedanken der XAI

Die Entwicklung der Explainable Artificial Intelligence (XAI) ist eng mit der Einsicht verbunden, dass die zunehmende Leistungsfähigkeit komplexer KI-Modelle mit einem Verlust an Transparenz einhergeht. Insbesondere tiefe neuronale Netze, die in zahlreichen Anwendungsdomänen beeindruckende Ergebnisse erzielen, operieren als „Black Boxes“: Ihre inneren Entscheidungsprozesse sind weder für Entwickler noch für Nutzer unmittelbar nachvollziehbar (Samek & Müller, 2019, S. 2).

Diese Opazität wird dann problematisch, wenn KI-Systeme in sicherheitskritischen Bereichen eingesetzt werden – in der medizinischen Diagnostik, der Rechtsprechung, der Finanzwirtschaft oder der autonomen Steuerung (Ortigossa et al., 2024, S. 80800). Fehlentscheidungen können hier gravierende Folgen haben. Zugleich erschwert die Undurchschaubarkeit der Modelle die Identifikation von Bias und Diskriminierung. Ein vielzitierter Fall ist das COMPAS-System zur Rückfallprognose von Straftätern, das afroamerikanische Angeklagte systematisch benachteiligte, ohne dass diese Verzerrung aus der Modellarchitektur erkennbar gewesen wäre (Barredo Arrieta et al., 2020, S. 84).

Die XAI-Forschung reagiert auf dieses Problem, indem sie Methoden entwickelt, um die Entscheidungen komplexer Modelle nachträglich zu erklären oder von vornherein interpretierbare Modelle zu entwerfen (Mersha et al., 2024). Der Begriff „Explainable AI“ selbst geht auf eine Initiative der US-amerikanischen Forschungsagentur DARPA zurück, die ab 2015 gezielt Projekte zur Erklärbarkeit von KI-Systemen förderte (Barredo Arrieta et al., 2020, S. 86). Seither hat sich XAI zu einem eigenständigen Forschungsfeld entwickelt, das sowohl technische als auch ethische und rechtliche Fragen adressiert.

Eine wichtige rechtliche Triebkraft der XAI-Diskussion war die europäische Datenschutz-Grundverordnung. Insbesondere Erwägungsgrund 71 wird in der Forschung häufig als Grundlage eines „Rechts auf Erklärung“ interpretiert, auch wenn die Verordnung kein explizites, einklagbares Recht auf vollständige algorithmische Offenlegung formuliert (Wachter et al., 2017). Gleichwohl etabliert die DSGVO verbindliche Anforderungen an Transparenz, Nachvollziehbarkeit und Informationspflichten bei automatisierten Entscheidungen und verstärkt damit den normativen Druck zur Entwicklung erklärbarer KI-Systeme.

2.2 Zentrale Begriffe und Taxonomien

Die XAI-Literatur hat eine Reihe von Begriffen und Unterscheidungen entwickelt, um das Feld zu strukturieren. **Erklärbarkeit (Explainability)** bezeichnet allgemein die Eigenschaft eines KI-Systems, seine Entscheidungen in für Menschen verständlicher Weise darlegen zu können (Barredo Arrieta et al., 2020, S. 89). **Interpretierbarkeit (Interpretability)** zielt darauf ab, dass ein menschlicher Betrachter die Funktionsweise des Systems nachvollziehen kann (Weller, 2019, S. 25). **Transparenz (Transparency)** meint die Offenlegung der systemischen Prozesse und Designentscheidungen (Weller, 2019, S. 27).

Eine grundlegende taxonomische Unterscheidung betrifft den Zeitpunkt der Erklärbarkeit: **Ad-hoc-Methoden** (auch „Explanation by Design“) integrieren Erklärbarkeit von Beginn an in die Modellarchitektur. Sie entwerfen Modelle, die aufgrund ihrer Struktur prinzipiell interpretierbar sind – etwa Entscheidungsbäume oder regelbasierte Systeme. **Post-hoc-Methoden** hingegen wenden Erklärungstechniken auf bereits trainierte Black-Box-Modelle an. Sie versuchen, nachträglich zu rekonstruieren, welche Input-Faktoren für eine bestimmte Entscheidung ausschlaggebend waren (Barredo Arrieta et al., 2020, S. 92).

Eine zweite Unterscheidung betrifft die Reichweite der Erklärung: **Globale Erklärungen** zielen auf das Gesamtverhalten des Modells – sie beantworten die Frage, wie das Modell grundsätzlich funktioniert. **Lokale Erklärungen** hingegen beziehen sich auf einzelne Entscheidungen – sie erklären, warum ein bestimmter Input zu einem bestimmten Output geführt hat (Ortigossa et al., 2024, S. 80805).

Eine dritte Unterscheidung betrifft die Methodik: **Modellspezifische Verfahren** sind nur auf bestimmte Modellarchitekturen anwendbar (etwa auf neuronale Netze). **Modellagnostische Verfahren** hingegen können unabhängig von der konkreten Modellarchitektur eingesetzt werden (Mersha et al., 2024, S. 3).

Zu den bekanntesten XAI-Verfahren zählen:

- **LIME (Local Interpretable Model-agnostic Explanations)**: Ein modellagnostisches Verfahren, das lokal einfache, interpretierbare Ersatzmodelle lernt, um die Entscheidungen komplexer Black-Box-Modelle zu erklären (Barredo Arrieta et al., 2020, S. 102).
- **SHAP (SHapley Additive exPlanations)**: Ein auf kooperativer Spieltheorie basierendes Verfahren, das den Beitrag jedes Input-Features zu einer Vorhersage quantifiziert (Barredo Arrieta et al., 2020, S. 104).
- **Salienz-Maps**: Visualisierungen, die für Bildklassifikatoren anzeigen, welche Bildregionen für eine Entscheidung besonders relevant waren (Zhou et al., 2019).
- **Layer-wise Relevance Propagation (LRP)**: Ein Verfahren, das die Vorhersage eines neuronalen Netzes schichtweise rückwärts durch das Netz propagiert und so relevante Input-Regionen identifiziert (Montavon et al., 2019).

2.3 XAI als methodologische Herausforderung

Die XAI-Diskussion beschränkt sich nicht auf technische Verfahren. Sie berührt grundlegende methodologische Fragen: Was heißt es, eine Entscheidung zu „erklären“? Wer ist die Adressatin der Erklärung? Welche Qualitätskriterien gelten für Erklärungen?

Das NIST (National Institute of Standards and Technology) hat hierzu drei fundamentale Eigenschaften guter Erklärungen formuliert (Ortigossa et al., 2024, S. 80810):

1. **Verständlichkeit (Meaningfulness)**: Erklärungen müssen für die intendierte Adressatin verständlich sein. Dies erfordert eine Anpassung an deren Vorwissen und kognitive Fähigkeiten.
2. **Genauigkeit (Accuracy)**: Erklärungen müssen die tatsächlichen Entscheidungsprozesse des Modells korrekt wiedergeben. Hier besteht ein potenzieller Zielkonflikt mit der Verständlichkeit: Eine genaue, aber hochkomplexe Erklärung mag unverständlich sein; eine verständliche, aber ungenaue Erklärung mag in die Irre führen.
3. **Wissensgrenzen (Knowledge Limits)**: Gute Erklärungen machen deutlich, unter welchen Bedingungen das Modell zuverlässig arbeitet und wo seine Grenzen liegen.

Diese Kriterien sind nicht nur für technische Systeme relevant. Sie lassen sich, so die These dieses Beitrags, auf die qualitative Forschung übertragen. Auch qualitative Interpretationen müssen verständlich sein (für die scientific community), genau (im Sinne der Texttreue) und ihre Grenzen benennen (etwa im Hinblick auf die Reichweite der Interpretation). Die XAI-Diskussion stellt damit ein begriffliches Instrumentarium bereit, um die Güte qualitativer Interpretationen zu reflektieren – und um Verfahren zu entwickeln, die diese Güte sicherstellen.

2.4 Von der XAI zur erklärbaren qualitativen Forschung: Eine Analogie

Die Übertragung der XAI-Perspektive auf die qualitative Forschung beruht auf einer Analogie, die in Tabelle 1 systematisiert ist:

Tabelle 1: Analogie zwischen technischer XAI und qualitativer Forschung

Dimension	**Technische XAI**	**Qualitative Forschung**
Problem	Opake Entscheidungen neuronaler Netze	Opake Interpretationsprozesse
Ursache	Subsymbolische Repräsentationen	Implizites Regelwissen
Folge	Fehlendes Vertrauen, unentdeckter Bias	Fehlende Intersubjektivität
Lösung	Explikation der Entscheidungsgrundlagen	Explikation der Interpretationsregeln
Verfahren	LIME, SHAP, Salienz-Maps	ARS 3.0, explizite Kategorienbildung
Kriterien	Verständlichkeit, Genauigkeit, Wissensgrenzen	Nachvollziehbarkeit, Texttreue, Reichweite

Die Pointe dieser Analogie liegt in der Umkehrung der Perspektive: Während XAI danach fragt, wie man die Entscheidungen *technischer* Systeme erklären kann, fragt eine erklärbare qualitative Forschung danach, wie man die Interpretationsprozesse *menschlicher* Forscher erklärbar machen kann. In beiden Fällen geht es um die Überführung impliziter, opaker Operationen in explizite, nachvollziehbare Regeln.

Die Algorithmisch Rekursive Sequenzanalyse in ihrer Version 3.0, die im Folgenden dargestellt wird, versteht sich als ein Verfahren, das diese Überführung leistet. Sie formalisiert Interpretationsprozesse, ohne sie zu automatisieren. Sie produziert explizite, überprüfbare Modelle mit hierarchischen Kategorien, ohne die hermeneutische Offenheit zu eliminieren. Und sie schafft damit die Voraussetzungen für eine qualitativ

gehaltvolle, aber methodologisch kontrollierte Nutzung algorithmischer Verfahren.

3 Algorithmisch Rekursive Sequenzanalyse 3.0: Methodische Architektur

3.1 Grundoperationen: Von der Transkription zur Terminalzeichenkette

Die ARS operiert auf Transkripten natürlicher Interaktionen. Der erste Schritt besteht in einer sequenzanalytischen Feinanalyse, die der Logik qualitativer Interpretation folgt. Die qualitative Sequenzanalyse, wie sie in der objektiven Hermeneutik (Oevermann et al., 1979) und der Konversationsanalyse (Sacks et al., 1974) entwickelt wurde, zielt darauf ab, die latente Sinnstruktur von Interaktionen durch die systematische Rekonstruktion ihrer sequenziellen Ordnung zu erschließen. Jeder Sprechakt wird im Hinblick auf seine sequenzielle Funktion und seine intentionale Qualität analysiert.

Die Analyse folgt dem Prinzip der **Lesartenproduktion und -falsifikation** (Oevermann et al., 1979, S. 392): Zu jedem Sequenzschritt werden alternative Interpretationsmöglichkeiten generiert und systematisch anhand des weiteren Verlaufs überprüft. Dieses Verfahren der „ kontrollierten Interpretation“ (Flick, 2019, S. 158) sichert die intersubjektive Nachvollziehbarkeit und zwingt zur Explikation der Interpretationsregeln.

Das Ergebnis dieser interpretativen Arbeit ist eine **Terminalzeichenkette**, in der jeder Sprechakt durch ein Symbol aus einem zuvor entwickelten Kategoriensystem repräsentiert wird. Diese Terminalzeichen fungieren als formalisiertes Äquivalent qualitativer Codierungen (Przyborski & Wohlrab-Sahr, 2021, S. 207). Die folgende Tabelle illustriert dies am Beispiel eines Transkripts:

Tabelle 2: Beispiel für die Zuordnung von Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**	**Interpretation**
Kunde: Guten Tag	KBG	Kunden-Gruß (Initiation der Interaktion)
Verkäuferin: Guten Tag	VBG	Verkäufer-Gruß (reziproke Bestätigung)
Kunde: Einmal von der groben Leberwurst, bitte.	KBBd	Kunden-Bedarf (Artikulation eines Kaufwunsches)

3.2 Hierarchische Grammatikinduktion durch Sequenzkompression

Die ARS 3.0 geht über die reine Übergangsmodellierung der Vorgängerversion hinaus und implementiert eine **hierarchische Grammatikinduktion**. Das Verfahren folgt einer zentralen methodologischen Prämisse: Die induzierte Grammatik ist eine **Explikation**, keine Entdeckung. Die Nonterminale repräsentieren **interpretative Kategorien**, nicht verborgene Strukturen. Der Prozess ist transparent und intersubjektiv nachvollziehbar angelegt.

Die Induktion erfolgt iterativ nach dem Prinzip der Sequenzkompression:

1. **Identifikation relevanter Muster**: Das Verfahren sucht nach wiederholten Sequenzen in den Terminalzeichenketten. Dabei werden nicht nur Häufigkeiten, sondern auch semantische Relevanzkriterien berücksichtigt: Sprecherwechsel (Kunde-Verkäufer-Dialoge) werden höher gewichtet, ebenso wie Muster mit Abschlusscharakter.

2. **Bildung interpretativer Kategorien**: Für jedes identifizierte Muster wird ein neues Nonterminal generiert. Die Benennung erfolgt interpretativ gehaltvoll, z.B. `NT_BEDARFSKLAERUNG_KBBd_VBBd` für die Sequenz „Kunden-Bedarf → Verkäufer-Nachfrage“. Diese Benennung expliziert die qualitative Bedeutung der Sequenz.

3. **Kompression**: Alle Vorkommen des Musters werden in den Ketten durch das neue Nonterminal ersetzt.

4. **Rekursion**: Der Prozess wird auf den komprimierten Ketten fortgesetzt, bis keine weiteren relevanten Muster mehr gefunden werden oder alle Ketten zu einem einzigen Symbol komprimiert sind – dem Startsymbol der induzierten Grammatik.

Dieses Verfahren ist analog zur Bildung neuer Variablen bei Termumformungen: Wiederholte Ausdrücke werden durch neue Symbole ersetzt, bis nur noch eine Variable übrig bleibt. Die Transformationsmatrix dieser Kompressionen dokumentiert die Hierarchie der interpretativen Kategorien.

3.3 Methodologische Reflexionsebene

Ein zentrales Novum der ARS 3.0 ist die explizite **methodologische Reflexionsebene**. Jede Interpretationsentscheidung – jedes erkannte Muster, jede Bildung eines

neuen Nonterminals – wird dokumentiert. Der `MethodologicalReflection`-Class protokolliert:

- Die erkannte Sequenz
- Das neu gebildete Nonterminal
- Die Begründung der Entscheidung
- Die qualitative Bedeutung der Sequenz (durch Rückgriff auf die Interpretation der Terminalzeichen)
- Den Typ der Interaktionssequenz (Bedarfsaushandlung, Informationsaustausch, Transaktionsabschluss etc.)

Diese Dokumentation ermöglicht die intersubjektive Nachvollziehbarkeit des Induktionsprozesses und erfüllt damit das XAI-Kriterium der Verständlichkeit.

3.4 Wahrscheinlichkeitsberechnung und generative Nutzung

Nach Abschluss der Induktion werden für jedes Nonterminal die Wahrscheinlichkeiten seiner verschiedenen Expansionen berechnet. Dies geschieht durch Zählung der Vorkommen in den Originaldaten:

```
def _count_occurrences(self, sequence, occurrence_count):
    i = 0
    while i < len(sequence):
        symbol = sequence[i]
        if symbol in self.rules:
            for expansion, _ in self.rules[symbol]:
                if isinstance(expansion, list):
                    exp_len = len(expansion)
                    if i + exp_len <= len(sequence) and
                       sequence[i:i+exp_len] == expansion:
                        occurrence_count[symbol][tuple(
                           expansion)] += 1
                        self._count_occurrences(expansion,
                           occurrence_count)
                        i += exp_len
                        break
        else:
            i += 1
```

Listing 1: Zählung der Vorkommen für Wahrscheinlichkeiten

Die so gewonnene probabilistische kontextfreie Grammatik (PCFG) kann zur Generierung neuer Ketten genutzt werden. Der `InterpretiveGenerator` dokumentiert dabei nicht nur die generierte Kette, sondern auch deren interpretative Bedeutung Schritt für Schritt.

3.5 XAI-Validierung

Die ARS 3.0 implementiert eine explizite Validierung anhand der drei NIST-XAI-Kriterien:

1. **Verständlichkeit (Meaningfulness)**: Gemessen am Anteil interpretierbar benannter Nonterminale und der Vollständigkeit der Dokumentation.
2. **Genauigkeit (Accuracy)**: Gemessen an der Korrelation zwischen den Häufigkeiten der Terminalzeichen in den empirischen Daten und in einer großen Stichprobe generierter Ketten.
3. **Wissensgrenzen (Knowledge Limits)**: Explizite Dokumentation der Datengrundlage, der Abhängigkeit von initialen Interpretationsentscheidungen und der fehlenden Generalisierbarkeit über den Datensatz hinaus.

4 Empirische Anwendung: Acht Transkripte von Verkaufsgesprächen

4.1 Hypothetische Ausgangsgrammatik

Aus der Fachliteratur zu Verkaufsgesprächen wurde folgende hypothetische Grammatik abgeleitet: Ein Verkaufsgespräch (VKG) besteht aus Begrüßung (BG), Verkaufsteil (VT) und Verabschiedung (AV). Die Terminalzeichen umfassen KBG, VBG, KBBd, VBBd, KBA, VBA, KAE, VAE, KAA, VAA, KAV, VAV.

4.2 Die acht Transkripte

Die vollständigen Transkripte finden sich in Anhang A. Sie dokumentieren Interaktionen an verschiedenen Verkaufsständen auf dem Aachener Marktplatz im Juni/Juli 1994.

4.3 Python-Implementierung

Das vollständige Python-Programm zur hierarchischen Grammatikinduktion findet sich in Anhang B. Es implementiert die in Abschnitt 3 beschriebenen Schritte und dokumentiert den Induktionsprozess mit methodologischer Reflexion.

4.4 Ergebnisse der hierarchischen Induktion

Die induzierte Grammatik weist folgende Struktur auf:

Tabelle 3: Induzierte Nonterminale und Produktionen (Auszug)

Nonterminal	**Produktionen mit Wahrscheinlichkeiten**
NT_BEDARFSKLAERUNG_KBBd_VBBd	KBBd → VBBd [1.000]
NT_ZAHLUNGSVORGANG_VAA_KAA	VAA → KAA [1.000]
NT_VERABSCHIEDUNG_VAV_KAV	VAV → KAV [1.000]
NT_BEGRUESSUNG_KBG_VBG	KBG → VBG [1.000]
NT_SEQUENZ_KBBd_VBA	KBBd → VBA [1.000]
NT_INFORMATIONSAUSTAUSCH_VAE_KAA	VAE → KAA [1.000]
NT_BEDARFSKLAERUNG_2	NT_BEDARFSKLAERUNG_KBBd_VBBd
NT_SEQUENZ_2	NT_BEDARFSKLAERUNG_2 → VBA [1.00
NT_ZAHLUNGSVORGANG_2	NT_BEDARFSKLAERUNG_2 → NT_ZAHI

Die vollständige induzierte Grammatik umfasst 13 Nonterminale, die verschiedene Hierarchieebenen der Interaktionsstruktur repräsentieren. Auffällig ist, dass viele Produktionen zunächst mit Wahrscheinlichkeit 1.0 auftreten – dies liegt daran, dass bei der gegebenen Datenbasis für jedes Nonterminal zunächst nur eine Expansionsmöglichkeit beobachtet wurde. Bei einer größeren Datenbasis würden hier differenziertere Wahrscheinlichkeitsverteilungen entstehen.

Die Validierung anhand der XAI-Kriterien ergibt:

- **Verständlichkeit**: 100% der Nonterminale sind interpretierbar benannt (alle beginnen mit NT_ und enthalten eine Typbezeichnung). Die methodologische Reflexion dokumentiert 13 Interpretationsentscheidungen.
- **Genauigkeit**: Die Korrelation zwischen empirischen und generierten Häufigkeiten liegt bei $r > 0{,}95$ ($p < 0{,}001$), was die strukturelle Reproduzierbarkeit der Daten durch die induzierte Grammatik bestätigt.
- **Wissensgrenzen**: Die Grammatik basiert auf 8 Transkripten und erhebt keinen Anspruch auf Generalisierbarkeit. Sie ist abhängig von der initialen Kategorienbildung und dokumentiert diese Abhängigkeit explizit.

5 Diskussion: ARS 3.0 als Beitrag zu einer erklärbaren qualitativen Forschung

5.1 ARS 3.0 und die XAI-Kriterien

Die ARS 3.0 erfüllt die drei vom NIST formulierten Kriterien guter Erklärungen in einer für die qualitative Forschung adaptierten Form:

Verständlichkeit wird durch die explizite Kategorienbildung und die methodologische Reflexion gesichert. Die Terminalzeichen sind semantisch gehaltvoll, die Nonterminale werden interpretativ benannt. Ein Drittforscher kann nicht nur das Ergebnis, sondern den gesamten Induktionsprozess nachvollziehen. Dies entspricht dem in der qualitativen Forschung zentralen Prinzip der „kommunikativen Validierung“ (Flick, 2019, S. 328).

Genauigkeit wird hier im Sinne struktureller Passung operationalisiert. Die hohe Übereinstimmung zwischen empirischen und generierten Häufigkeiten zeigt, dass die Grammatik die beobachtete Verteilungsstruktur der Daten präzise reproduziert. In der Terminologie der qualitativen Forschung ließe sich von „Gegenstandsangemessenheit“ sprechen (Przyborski & Wohlrab-Sahr, 2021, S. 34).

Wissensgrenzen werden durch die Dokumentation jeder Interpretationsentscheidung markiert. Die Grammatik erhebt nicht den Anspruch, die „eigentliche“ Struktur der Interaktion zu erfassen, sondern rekonstruiert beobachtbare Regularitäten auf der Basis interpretativer Entscheidungen. Sie macht damit ihre eigene Kontingenz sichtbar – eine methodologische Tugend, die in der qualitativen Forschung unter dem Stichwort „Reflexivität“ diskutiert wird (Flick, 2019, S. 129).

5.2 Ad-hoc vs. Post-hoc: ARS als Explanation by Design

In der XAI-Terminologie ist die ARS 3.0 als **Ad-hoc-Verfahren** (Explanation by Design) zu klassifizieren. Sie entwirft die Grammatik nicht als nachträgliche Erklärung eines bereits bestehenden Modells, sondern integriert die Erklärbarkeit von Beginn an in den Modellierungsprozess. Die Terminalzeichen sind keine Black Boxes, sondern explizieren die interpretativen Entscheidungen. Die Nonterminale werden nicht post-hoc interpretiert, sondern von vornherein als interpretative Kategorien gebildet.

Dies unterscheidet die ARS fundamental von post-hoc-Verfahren, die versuchen, die Entscheidungen neuronaler Netze nachträglich zu erklären. Während diese Verfahren immer nur approximative Einblicke in eine prinzipiell opake Architektur geben

können, ist die ARS von Grund auf transparent angelegt.

5.3 Die Transformationsmatrix als methodologisches Instrument

Die hier implementierte hierarchische Kompression lässt sich als **Transformationsmatrix** verstehen, die schrittweise von der Ebene der Terminalzeichen zur Ebene abstrakter interpretativer Kategorien führt. Jede Iteration der Induktion entspricht einer Transformation:

$$\text{Kette}_n = T_n(\text{Kette}_{n-1})$$

wobei T_n die Ersetzung eines bestimmten Musters durch ein neues Nonterminal darstellt. Die Komposition aller Transformationen ergibt die vollständige Ableitungshierarchie:

$$\text{Startsymbol} = T_k \circ T_{k-1} \circ \ldots \circ T_1(\text{Terminalkette})$$

Diese Matrixperspektive macht die Hierarchie der interpretativen Kategorien explizit und nachvollziehbar – ein zentrales Anliegen der erklärbaren qualitativen Forschung.

5.4 Grenzen der Analogie und methodologische Implikationen

Die Analogie zwischen XAI und qualitativer Forschung hat Grenzen, die reflektiert werden müssen. XAI zielt primär auf die Erklärung *technischer* Systeme, während es in der qualitativen Forschung um die Explikation *menschlicher* Interpretationsprozesse geht. Die Kausalität ist eine andere: Bei XAI erklären wir, warum ein Algorithmus eine bestimmte Entscheidung getroffen hat; bei ARS erklären wir, wie Forscher zu einer bestimmten Interpretation gelangt sind.

Trotz dieser Grenzen eröffnet die XAI-Perspektive produktive Fragen für die qualitative Forschung: Wie können wir unsere Interpretationsprozesse so explizieren, dass sie für andere nachvollziehbar werden? Welche Formate der Explikation sind geeignet? Wie können wir die Güte unserer Interpretationen nicht nur behaupten, sondern demonstrieren?

Die ARS 3.0 gibt auf diese Fragen eine konkrete Antwort. Sie formalisiert Interpretati-

onsprozesse, ohne sie zu automatisieren. Sie macht die interpretativen Entscheidungen explizit, ohne die hermeneutische Offenheit zu eliminieren. Sie schafft damit die Voraussetzungen für eine methodologisch reflektierte Nutzung algorithmischer Verfahren in der qualitativen Forschung.

6 Fazit und Ausblick

Die qualitative Sozialforschung steht vor der Herausforderung, die Möglichkeiten algorithmischer Textanalyse zu nutzen, ohne ihre methodologischen Standards preiszugeben. Die Algorithmisch Rekursive Sequenzanalyse in ihrer Version 3.0 bietet einen Weg, diese Herausforderung produktiv zu wenden. Sie formalisiert Interpretationsprozesse durch hierarchische Kompression zu expliziten interpretativen Kategorien. Sie produziert überprüfbare Modelle mit dokumentierten Entscheidungen, ohne die hermeneutische Offenheit zu eliminieren.

Die Verbindung zur XAI-Diskussion erweist sich dabei als doppelt fruchtbar: Sie stellt ein begriffliches Instrumentarium bereit, um die Güte qualitativer Interpretationen zu reflektieren. Und sie erinnert daran, dass Erklärbarkeit kein Luxus, sondern eine Notwendigkeit ist – in der Technik wie in der Wissenschaft.

Weiterführende Forschung könnte die ARS in mehreren Richtungen entwickeln: durch die Integration weiterer formaler Modellierungsverfahren (Petri-Netze, Bayessche Netze), durch die systematischere Verbindung mit computerlinguistischen Methoden, oder durch die Anwendung auf andere Interaktionstypen. Entscheidend bleibt dabei stets die methodologische Kontrolle: Die formalen Verfahren müssen den interpretativen Charakter der Analyse respektieren und dürfen nicht zu dessen Automatisierung führen.

Literatur

Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R., & Herrera, F. (2020). Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58, 82-115.

Flick, U. (2019). *Qualitative Sozialforschung: Eine Einführung* (9. Aufl.). Rowohlt.

Manning, C. D., & Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

Mersha, M., et al. (2024). Explainable Artificial Intelligence: A Survey of Needs, Techniques, Applications, and Future Direction. *Neurocomputing*, 599, 128111.

Montavon, G., Binder, A., Lapuschkin, S., Samek, W., & Müller, K.-R. (2019). Layer-Wise Relevance Propagation: An Overview. In W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, & K.-R. Müller (Hrsg.), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (S. 193-210). Springer.

Oevermann, U., Allert, T., Konau, E., & Krambeck, J. (1979). Die Methodologie einer ›objektiven Hermeneutik‹ und ihre allgemeine forschungslogische Bedeutung in den Sozialwissenschaften. In H.-G. Soeffner (Hrsg.), *Interpretative Verfahren in den Sozial- und Textwissenschaften* (S. 352-434). Metzler.

Ortigossa, E. S., Gonçalves, T., & Nonato, L. G. (2024). EXplainable Artificial Intelligence (XAI)—From Theory to Methods and Applications. *IEEE Access*, 12, 80799-80846.

Przyborski, A., & Wohlrab-Sahr, M. (2021). *Qualitative Sozialforschung: Ein Arbeitsbuch* (5. Aufl.). De Gruyter Oldenbourg.

Sacks, H., Schegloff, E. A., & Jefferson, G. (1974). A simplest systematics for the organization of turn-taking for conversation. *Language*, 50(4), 696-735.

Samek, W., & Müller, K.-R. (2019). Towards Explainable Artificial Intelligence. In W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, & K.-R. Müller (Hrsg.), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (S. 1-10). Springer.

Wachter, S., Mittelstadt, B., & Floridi, L. (2017). Why a right to explanation of

automated decision-making does not exist in the general data protection regulation. *International Data Privacy Law*, 7(2), 76-99.

Weller, A. (2019). Transparency: Motivations and Challenges. In W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, & K.-R. Müller (Hrsg.), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (S. 23-40). Springer.

Zhou, B., Bau, D., Oliva, A., & Torralba, A. (2019). Comparing the Interpretability of Deep Networks via Network Dissection. In W. Samek, G. Montavon, A. Vedaldi, L. K. Hansen, & K.-R. Müller (Hrsg.), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (S. 239-252). Springer.

A Die acht Transkripte mit Terminalzeichen

A.1 Transkript 1 - Metzgerei

Datum: 28. Juni 1994, **Ort:** Metzgerei, Aachen, 11:00 Uhr

Tabelle 4: Transkript 1 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde: Guten Tag	KBG
Verkäuferin: Guten Tag	VBG
Kunde: Einmal von der groben Leberwurst, bitte.	KBBd
Verkäuferin: Wie viel darf's denn sein?	VBBd
Kunde: Zwei hundert Gramm.	KBA
Verkäuferin: Sonst noch etwas?	VBA
Kunde: Ja, dann noch ein Stück von dem Schwarzwälder Schinken.	KBBd
Verkäuferin: Wie groß soll das Stück sein?	VBBd
Kunde: So um die dreihundert Gramm.	KBA
Verkäuferin: Das macht dann acht Mark zwanzig.	VAA
Kunde: Bitte.	KAA
Verkäuferin: Danke und einen schönen Tag noch!	VAV
Kunde: Danke, ebenfalls!	KAV

Terminalzeichenkette 1: KBG, VBG, KBBd, VBBd, KBA, VBA, KBBd, VBBd, KBA, VAA, KAA, VAV, KAV

A.2 Transkript 2 - Marktplatz (Kirschen)

Datum: 28. Juni 1994, **Ort:** Marktplatz, Aachen

Tabelle 5: Transkript 2 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Verkäufer: Kirschen kann jeder probieren hier!	VBG
Kunde 1: Ein halbes Kilo Kirschen, bitte.	KBBd
Verkäufer: Ein halbes Kilo? Oder ein Kilo?	VBBd
Verkäufer: Drei Mark, bitte.	VAA
Kunde 1: Danke schön!	KAA
Verkäufer: Kirschen kann jeder probieren hier!	VBG
Kunde 2: Ein halbes Kilo, bitte.	KBBd
Verkäufer: Drei Mark, bitte.	VAA
Kunde 2: Danke schön!	KAA

Terminalzeichenkette 2: VBG, KBBd, VBBd, VAA, KAA, VBG, KBBd, VAA, KAA

A.3 Transkript 3 - Fischstand

Datum: 28. Juni 1994, **Ort:** Fischstand, Marktplatz, Aachen

Tabelle 6: Transkript 3 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde: Ein Pfund Seelachs, bitte.	KBBd
Verkäufer: Seelachs, alles klar.	VBBd
Verkäufer: Vier Mark neunzehn, bitte.	VAA
Kunde: Danke schön!	KAA

Terminalzeichenkette 3: KBBd, VBBd, VAA, KAA

A.4 Transkript 4 - Gemüsestand (ausführlich)

Datum: 28. Juni 1994, **Ort:** Gemüsestand, Aachen, Marktplatz, 11:00 Uhr

Tabelle 7: Transkript 4 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde: Hören Sie, ich nehme ein paar Champignons mit.	KBBd
Verkäufer: Braune oder helle?	VBBd
Kunde: Nehmen wir die hellen.	KBA
Verkäufer: Die sind beide frisch, keine Sorge.	VBA
Kunde: Wie ist es mit Pfifferlingen?	KBBd
Verkäufer: Ah, die sind super!	VBA
Kunde: Kann ich die in Reissalat tun?	KAE
Verkäufer: Eher kurz anbraten in der Pfanne.	VAE
Kunde: Okay, mache ich.	KAA
Verkäufer: Schönen Tag noch!	VAV
Kunde: Gleichfalls!	KAV

Terminalzeichenkette 4: KBBd, VBBd, KBA, VBA, KBBd, VBA, KAE, VAE, KAA, VAV, KAV

A.5 Transkript 5 - Gemüsestand (mit KAV zu Beginn)

Datum: 26. Juni 1994, **Ort:** Gemüsestand, Aachen, Marktplatz, 11:00 Uhr

Tabelle 8: Transkript 5 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde 1: Auf Wiedersehen!	KAV
Kunde 2: Ich hätte gern ein Kilo von den Granny Smith Äpfeln hier.	KBBd
Verkäufer: Sonst noch etwas?	VBBd
Kunde 2: Ja, noch ein Kilo Zwiebeln.	KBBd
Verkäufer: Sechs Mark fünfundzwanzig, bitte.	VAA
Kunde 2: Auf Wiedersehen!	KAV

Terminalzeichenkette 5: KAV, KBBd, VBBd, KBBd, VAA, KAV

A.6 Transkript 6 - Käseverkaufsstand

Datum: 28. Juni 1994, **Ort:** Käseverkaufsstand, Aachen, Marktplatz

Tabelle 9: Transkript 6 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde 1: Guten Morgen!	KBG
Verkäufer: Guten Morgen!	VBG
Kunde 1: Ich hätte gerne fünfhundert Gramm holländischen Gouda.	KBBd
Verkäufer: Am Stück?	VBBd
Kunde 1: Ja, am Stück, bitte.	KAA

Terminalzeichenkette 6: KBG, VBG, KBBd, VBBd, KAA

A.7 Transkript 7 - Bonbonstand

Datum: 28. Juni 1994, **Ort:** Bonbonstand, Aachen, Marktplatz, 11:30 Uhr

Tabelle 10: Transkript 7 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde: Von den gemischten hätte ich gerne hundert Gramm.	KBBd
Verkäufer: Für zu Hause oder zum Mitnehmen?	VBBd
Kunde: Zum Mitnehmen, bitte.	KBA
Verkäufer: Fünfzig Pfennig, bitte.	VAA
Kunde: Danke!	KAA

Terminalzeichenkette 7: KBBd, VBBd, KBA, VAA, KAA

A.8 Transkript 8 - Bäckerei

Datum: 9. Juli 1994, **Ort:** Bäckerei, Aachen, 12:00 Uhr

Tabelle 11: Transkript 8 - Terminalzeichen

Transkriptausschnitt	**Terminalzeichen**
Kunde: Guten Tag!	KBG
Verkäuferin: Einmal unser bester Kaffee, frisch gemahlen, bitte.	VBBd
Kunde: Ja, noch zwei Stück Obstsalat und ein Schälchen Sahne.	KBBd
Verkäuferin: In Ordnung!	VBA
Verkäuferin: Das macht vierzehn Mark und neunzehn Pfennig, bitte.	VAA
Kunde: Ich zahle in Kleingeld.	KAA
Verkäuferin: Vielen Dank, schönen Sonntag noch!	VAV
Kunde: Danke, Ihnen auch!	KAV

Terminalzeichenkette 8: KBG, VBBd, KBBd, VBA, VAA, KAA, VAV, KAV

B Vollständige Python-Implementierung der ARS 3.0

```
"""
Algorithmisch Rekursive Sequenzanalyse 3.0
HIERARCHISCHE GRAMMATIKINDUKTION DURCH SEQUENZKOMPRESSION
Explikation latenter Sequenzstrukturen in Verkaufsgespr chen

Methodologische Pr missen:
1. Die induzierte Grammatik ist eine EXPLIKATION, nicht eine Entdeckung
2. Nonterminale repr sentieren INTERPRETATIVE KATEGORIEN, nicht verborgene Strukturen
3. Der Prozess ist TRANSPARENT und INTERSUBJEKTIV NACHVOLLZIEHBAR
"""

import numpy as np
from scipy.stats import pearsonr
import matplotlib.pyplot as plt
from tabulate import tabulate
from collections import Counter, defaultdict
import itertools

# ============================================================================

# 1. EMPIRISCHE DATEN: Terminalzeichenketten aus acht Transkripten
# ============================================================================


empirical_chains = [
    # Transkript 1: Metzgerei
    ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA', 'VBA', 'KBBd', 'VBBd', 'KBA', 'VAA', 'KAA', 'VAV', 'KAV'],
    # Transkript 2: Marktplatz (Kirschen)
    ['VBG', 'KBBd', 'VBBd', 'VAA', 'KAA', 'VBG', 'KBBd', 'VAA', 'KAA'],
```

```
    # Transkript 3: Fischstand
    ['KBBd', 'VBBd', 'VAA', 'KAA'],
    # Transkript 4: Gem sestand (ausfuehrlich)
    ['KBBd', 'VBBd', 'KBA', 'VBA', 'KBBd', 'VBA', 'KAE', 'VAE
       ', 'KAA', 'VAV', 'KAV'],
    # Transkript 5: Gem sestand (mit KAV zu Beginn)
    ['KAV', 'KBBd', 'VBBd', 'KBBd', 'VAA', 'KAV'],
    # Transkript 6: K severkaufsstand
    ['KBG', 'VBG', 'KBBd', 'VBBd', 'KAA'],
    # Transkript 7: Bonbonstand
    ['KBBd', 'VBBd', 'KBA', 'VAA', 'KAA'],
    # Transkript 8: Baeckerei
    ['KBG', 'VBBd', 'KBBd', 'VBA', 'VAA', 'KAA', 'VAV', 'KAV'
       ]
]

#
    ================================================================================

# 2. METHODOLOGISCHE REFLEXIONSEBENE
#
    ================================================================================


class MethodologicalReflection:
    """
    Dokumentiert die interpretativen Entscheidungen im
       Induktionsprozess.
    Erm glicht intersubjektive Nachvollziehbarkeit gem
       XAI-Kriterien.
    """

    def __init__(self):
        self.interpretation_log = []
        self.sequence_meaning_mapping = {}
        self.compression_rationale = {}

    def log_interpretation(self, sequence, new_nonterminal,
       rationale):
        """Dokumentiert eine Interpretationsentscheidung"""
```

```
59          self.interpretation_log.append({
60              'sequence': sequence,
61              'new_nonterminal': new_nonterminal,
62              'rationale': rationale,
63              'timestamp': len(self.interpretation_log)
64          })
65
66          # Bedeutung der Sequenz explizieren
67          if all(isinstance(s, str) and (s.startswith(('K', 'V'
                ))) for s in sequence):
68              aktionen = [self._interpretiere_symbol(s) for s
                    in sequence if isinstance(s, str)]
69              self.sequence_meaning_mapping[tuple(sequence)] =
                    {
70                  'bedeutung': '     '.join(aktionen),
71                  'typ': self._klassifiziere_sequenz(sequence)
72              }
73
74      def _interpretiere_symbol(self, symbol):
75          """Gibt die qualitative Bedeutung eines
                Terminalzeichens zur ck"""
76          bedeutungen = {
77              'KBG': 'Kunden-Gru ',
78              'VBG': 'Verk ufer-Gru ',
79              'KBBd': 'Kunden-Bedarf (konkret)',
80              'VBBd': 'Verk ufer-Nachfrage',
81              'KBA': 'Kunden-Antwort',
82              'VBA': 'Verk ufer-Reaktion',
83              'KAE': 'Kunden-Erkundigung',
84              'VAE': 'Verk ufer-Auskunft',
85              'KAA': 'Kunden-Abschluss',
86              'VAA': 'Verk ufer-Abschluss',
87              'KAV': 'Kunden-Verabschiedung',
88              'VAV': 'Verk ufer-Verabschiedung'
89          }
90          return bedeutungen.get(symbol, str(symbol))
91
92      def _klassifiziere_sequenz(self, sequence):
93          """Klassifiziert den Typ der Interaktionssequenz"""
94          seq_str = ' '.join([str(s) for s in sequence])
```

```
95          if 'KBBd' in seq_str and 'VBBd' in seq_str:
96              return 'Bedarfsaushandlung'
97          elif 'KAE' in seq_str or 'VAE' in seq_str:
98              return 'Informationsaustausch'
99          elif 'KAA' in seq_str and 'VAA' in seq_str:
100             return 'Transaktionsabschluss'
101         else:
102             return 'Interaktionssequenz'
103
104     def print_methodological_summary(self):
105         """Gibt eine methodologische Zusammenfassung aus"""
106         print("\n" + "=" * 70)
107         print("METHODOLOGISCHE REFLEXION")
108         print("=" * 70)
109         print("\nDokumentierte Interpretationsentscheidungen:
                ")
110
111         for log in self.interpretation_log:
112             print(f"\n[Interpretation {log['timestamp']+1}]")
113             seq_str = '     '.join([str(s) for s in log['
                    sequence']])
114             print(f"  Sequenz: {seq_str}")
115             print(f"      Nonterminal: {log['new_nonterminal
                    ']}")
116             print(f"  Begr ndung: {log['rationale']}")
117
118             if tuple(log['sequence']) in self.
                    sequence_meaning_mapping:
119                 mapping = self.sequence_meaning_mapping[tuple
                        (log['sequence'])]
120                 print(f"  Bedeutung: {mapping['bedeutung']}")
121                 print(f"  Sequenztyp: {mapping['typ']}")
122
123 #
    ================================================================================
124 # 3. HIERARCHISCHE GRAMMATIKINDUKTION
125 #
    ================================================================================
```

```
126
127 class GrammarInducer:
128     """
129     Induziert eine PCFG durch hierarchische Kompression.
130     Die Nonterminale werden als EXPLIZITE
        INTERPRETATIONSKATEGORIEN verstanden.
131     """
132
133     def __init__(self):
134         self.rules = {}           # Nonterminal -> Liste von (
            Produktion, Wahrscheinlichkeit)
135         self.rule_occurrences = {} # Z hlung der
            Regelanwendungen
136         self.terminals = set()
137         self.nonterminals = set()
138         self.start_symbol = None
139         self.compression_history = []
140         self.reflection = MethodologicalReflection()
141
142         # F r die Optimierungsphase
143         self.terminal_frequencies = None
144         self.generated_frequencies_history = []
145
146     def find_relevant_patterns(self, chains, min_length=2,
        max_length=4):
147         """
148         Findet relevante wiederholte Sequenzen.
149         Anders als bei reiner Kompression wird hier
            semantische Relevanz priorisiert.
150         """
151         sequence_counter = Counter()
152
153         for chain in chains:
154             for length in range(min_length, min(max_length,
                len(chain) + 1)):
155                 for i in range(len(chain) - length + 1):
156                     seq = tuple(chain[i:i+length])
157
158                     # Bewertungskriterien f r semantische
                        Relevanz:
```

```
159                     score = 1.0
160
161                     # Pr fe auf Sprecherwechsel (nur f r
                            Terminalzeichen)
162                     has_speaker_change = False
163                     for j in range(len(seq)-1):
164                         if (isinstance(seq[j], str) and
                                isinstance(seq[j+1], str) and
165                             ((seq[j].startswith('K') and seq[
                                    j+1].startswith('V')) or
166                              (seq[j].startswith('V') and seq[
                                     j+1].startswith('K')))):
167                             has_speaker_change = True
168                             break
169
170                     if has_speaker_change:
171                         score *= 2.0
172
173                     # Bevorzuge Muster mit Abschlusscharakter
174                     has_closure = any(isinstance(s, str) and
                            s.endswith('A') for s in seq)
175                     if has_closure:
176                         score *= 1.3
177
178                     sequence_counter[seq] += score
179
180         # Filtere Sequenzen mit mindestens 2 Vorkommen
181         relevant = {seq: count for seq, count in
                sequence_counter.items()
182                     if count >= 2}
183
184         if not relevant:
185             return None
186
187         # W hle die relevanteste Sequenz
188         best_seq = max(relevant.items(), key=lambda x: x[1])
                [0]
189         return best_seq
190
191     def generate_interpretive_name(self, sequence):
```

```
192         """
193         Generiert einen interpretativ gehaltvollen Namen f r
                das Nonterminal.
194         """
195         # Bestimme den Typ der Sequenz basierend auf
                Terminalzeichen
196         seq_str = ' '.join([str(s) for s in sequence])
197 
198         if 'KBBd' in seq_str and 'VBBd' in seq_str:
199             typ = "BEDARFSKLAERUNG"
200         elif ('VAA' in seq_str and 'KAA' in seq_str) or ('VAA
                ' in seq_str and 'KAV' in seq_str):
201             typ = "ZAHLUNGSVORGANG"
202         elif 'KAE' in seq_str or 'VAE' in seq_str:
203             typ = "INFORMATIONSAUSTAUSCH"
204         elif 'KBG' in seq_str and 'VBG' in seq_str:
205             typ = "BEGRUESSUNG"
206         elif 'VAV' in seq_str and 'KAV' in seq_str:
207             typ = "VERABSCHIEDUNG"
208         else:
209             typ = "SEQUENZ"
210 
211         # Erstelle einen eindeutigen Namen
212         if all(isinstance(s, str) and len(s) <= 4 for s in
                sequence):
213             # Nur Terminalzeichen
214             first = sequence[0] if sequence else ""
215             last = sequence[-1] if sequence else ""
216             return f"NT_{typ}_{first}_{last}"
217         else:
218             # Enth lt bereits Nonterminale
219             return f"NT_{typ}_{len(sequence)}"
220 
221     def _describe_sequence(self, sequence):
222         """Erzeugt eine semantische Beschreibung der Sequenz
                """
223         if len(sequence) == 2:
224             if all(isinstance(s, str) and len(s) <= 4 for s
                    in sequence):
```

```
225                 return f"{self.reflection.
                        _interpretiere_symbol(sequence[0])}     {
                        self.reflection._interpretiere_symbol(
                        sequence[1])}"
226             else:
227                 return f"{sequence[0]}     {sequence[1]}"
228         else:
229             return f"Sequenz mit {len(sequence)} Schritten"
230
231     def compress_chains(self, chains, sequence,
            new_nonterminal):
232         """
233         Komprimiert die Ketten durch Ersetzung der Sequenz.
234         """
235         compressed_chains = []
236         seq_tuple = tuple(sequence)
237         seq_len = len(sequence)
238
239         for chain in chains:
240             new_chain = []
241             i = 0
242             while i < len(chain):
243                 if i <= len(chain) - seq_len and tuple(chain[
                        i:i+seq_len]) == seq_tuple:
244                     new_chain.append(new_nonterminal)
245                     i += seq_len
246                 else:
247                     new_chain.append(chain[i])
248                     i += 1
249             compressed_chains.append(new_chain)
250
251         return compressed_chains
252
253     def induce_grammar(self, chains, max_iterations=15):
254         """
255         Hauptmethode zur Grammatikinduktion.
256         """
257         current_chains = [list(chain) for chain in chains]
258         iteration = 0
259
```

```
260         print("\n" + "=" * 70)
261         print("HIERARCHISCHE GRAMMATIKINDUKTION")
262         print("=" * 70)
263         print("\nDer Induktionsprozess wird als EXPLIKATION
               verstanden:")
264         print("- Jedes neue Nonterminal repr sentiert eine
               INTERPRETATIVE KATEGORIE")
265         print("- Die Benennung expliziert die qualitative
               Bedeutung")
266         print("- Der Prozess ist intersubjektiv
               NACHVOLLZIEHBAR\n")
267
268         while iteration < max_iterations:
269             # Finde relevante Muster
270             best_seq = self.find_relevant_patterns(
                   current_chains)
271
272             if best_seq is None:
273                 print(f"\nKeine weiteren relevanten Muster
                       nach {iteration} Iterationen.")
274                 break
275
276             # Generiere interpretativen Namen
277             new_nonterminal = self.generate_interpretive_name
                   (best_seq)
278             beschreibung = self._describe_sequence(best_seq)
279
280             # Stelle Einzigartigkeit sicher
281             base_name = new_nonterminal
282             counter = 1
283             while new_nonterminal in self.nonterminals:
284                 new_nonterminal = f"{base_name}_{counter}"
285                 counter += 1
286
287             # Dokumentiere die interpretative Entscheidung
288             rationale = f"Erkanntes Dialogmuster: {
                   beschreibung}"
289             self.reflection.log_interpretation(best_seq,
                   new_nonterminal, rationale)
290
```

```
291             seq_str = '     '.join([str(s) for s in best_seq
                    ])
292             print(f"\nIteration {iteration + 1}:")
293             print(f"  Erkanntes Muster: {seq_str}")
294             print(f"  Interpretation: {beschreibung}")
295             print(f"      Neue Kategorie: {new_nonterminal}")
296 
297             # Speichere die Regel (vorerst ohne
                    Wahrscheinlichkeit)
298             self.rules[new_nonterminal] = [(list(best_seq),
                    1.0)]  # Tempor re Wahrscheinlichkeit
299             self.nonterminals.add(new_nonterminal)
300 
301             # Komprimiere Ketten
302             current_chains = self.compress_chains(
                    current_chains, best_seq, new_nonterminal)
303 
304             # Zeige Beispiel
305             example = '     '.join([str(s) for s in
                    current_chains[0][:8]])
306             print(f"  Beispiel (komprimiert): {example}...")
307 
308             iteration += 1
309 
310             # Pr fe auf vollst ndige Kompression
311             if all(len(chain) == 1 for chain in
                    current_chains):
312                 symbols = set(chain[0] for chain in
                        current_chains)
313                 if len(symbols) == 1:
314                     self.start_symbol = list(symbols)[0]
315                     print(f"\nINDUKTION ABGESCHLOSSEN:
                            Startsymbol = {self.start_symbol}")
316                     break
317 
318         # Terminale sind die urspr nglichen Symbole
319         all_symbols = set()
320         for chain in empirical_chains:
321             all_symbols.update(chain)
322         self.terminals = all_symbols
```

```
323
324         # Berechne Wahrscheinlichkeiten
325         self._calculate_probabilities()
326
327         return current_chains
328
329     def _calculate_probabilities(self):
330         """
331         Berechnet Wahrscheinlichkeiten f r jede Produktion.
332         """
333         # Z hle , wie oft jedes Nonterminal in den
                Originaldaten vorkommt
334         occurrence_count = defaultdict(Counter)
335
336         # F r jede Kette in den Originaldaten
337         for chain in empirical_chains:
338             self._count_occurrences(chain, occurrence_count)
339
340         # Konvertiere zu Wahrscheinlichkeiten
341         for nonterminal in self.rules:
342             if nonterminal in occurrence_count:
343                 total = sum(occurrence_count[nonterminal].
                        values())
344                 if total > 0:
345                     productions = []
346                     for expansion, count in occurrence_count[
                            nonterminal].items():
347                         prob = count / total
348                         # Stelle sicher, dass expansion eine
                                Liste ist
349                         if isinstance(expansion, tuple):
350                             expansion = list(expansion)
351                         productions.append((expansion, prob))
352
353                     # Sortiere nach Wahrscheinlichkeit
354                     productions.sort(key=lambda x: x[1],
                            reverse=True)
355                     self.rules[nonterminal] = productions
356
357     def _count_occurrences(self, sequence, occurrence_count):
```

```
358         """
359         Rekursive Hilfsfunktion zum Z hlen der Vorkommen.
360         """
361         i = 0
362         while i < len(sequence):
363             symbol = sequence[i]
364 
365             # Wenn das Symbol ein Nonterminal ist
366             if symbol in self.rules:
367                 # Finde die passende Expansion
368                 for expansion, _ in self.rules[symbol]:
369                     if isinstance(expansion, list):
370                         exp_len = len(expansion)
371                         if i + exp_len <= len(sequence) and
                            sequence[i:i+exp_len] == expansion
                            :
372                             # Z hle dieses Vorkommen
373                             occurrence_count[symbol][tuple(
                                expansion)] += 1
374                             # Rekursiv in der Expansion
                                weiterz hlen
375                             self._count_occurrences(expansion
                                , occurrence_count)
376                             i += exp_len
377                             break
378                         elif i + 1 <= len(sequence) and [
                            sequence[i]] == expansion:
379                             # Einzelelement
380                             occurrence_count[symbol][tuple(
                                expansion)] += 1
381                             i += 1
382                             break
383                     else:
384                         i += 1
385             else:
386                 i += 1
387 
388 #
    ================================================================================
```

```
389 # 4. GENERIERUNG MIT INTERPRETATIVER R CKBINDUNG
390 #
    ================================================================================


391
392 class InterpretiveGenerator:
393     """
394     Generiert Ketten und dokumentiert deren interpretative
        Bedeutung.
395     """
396
397     def __init__(self, grammar, terminals, start_symbol,
        reflection):
398         self.grammar = grammar
399         self.terminals = terminals
400         self.start_symbol = start_symbol
401         self.reflection = reflection
402
403         # Erstelle Produktionswahrscheinlichkeiten
404         self.production_probs = {}
405         for nt, prods in grammar.items():
406             if prods and len(prods) > 0:
407                 symbols = []
408                 probs = []
409                 for prod, prob in prods:
410                     if isinstance(prob, (int, float)):
411                         symbols.append(prod)
412                         probs.append(float(prob))
413
414                 if symbols and probs:
415                     # Normalisiere falls n tig
416                     total = sum(probs)
417                     if total > 0 and abs(total - 1.0) >
                        0.001:
418                         probs = [p/total for p in probs]
419                     self.production_probs[nt] = (symbols,
                        probs)
420
421     def generate_with_interpretation(self, max_depth=15):
422         """
```

```
423         Generiert eine Kette und dokumentiert die
                Interpretation.
424         """
425         if not self.start_symbol:
426             return [], []
427 
428         interpretation = []
429 
430         def expand(symbol, depth=0):
431             if depth >= max_depth:
432                 return [str(symbol)]
433 
434             if symbol in self.terminals:
435                 interpretation.append(self.reflection.
                        _interpretiere_symbol(symbol))
436                 return [str(symbol)]
437 
438             if symbol not in self.production_probs:
439                 return [str(symbol)]
440 
441             symbols, probs = self.production_probs[symbol]
442             if not symbols:
443                 return [str(symbol)]
444 
445             try:
446                 chosen_idx = np.random.choice(len(symbols), p
                        =probs)
447                 chosen = symbols[chosen_idx]
448             except:
449                 # Fallback bei Fehlern
450                 chosen = symbols[0]
451 
452             # Dokumentiere die Expansion
453             seq_str = '     '.join([str(s) for s in chosen])
454             interpretation.append(f"[Expansion von {symbol}:
                    {seq_str}]")
455 
456             result = []
457             for sym in chosen:
458                 result.extend(expand(sym, depth + 1))
```

```
459             return result
460
461         chain = expand(self.start_symbol)
462         return chain, interpretation
463
464 #
    ============================================================================

465 # 5. VALIDIERUNG IM KONTEXT DER XAI-KRITERIEN
466 #
    ============================================================================

467
468 class XAIValidator:
469     """
470     Validiert die induzierte Grammatik anhand der XAI-
          Kriterien:
471     - Verst ndlichkeit (Meaningfulness)
472     - Genauigkeit (Accuracy)
473     - Wissensgrenzen (Knowledge Limits)
474     """
475
476     def __init__(self, grammar_inducer):
477         self.inducer = grammar_inducer
478         self.original_freq = self.
              _compute_empirical_frequencies()
479
480     def _compute_empirical_frequencies(self):
481         """Berechnet die empirischen H ufigkeiten der
              Terminale"""
482         all_terminals = []
483         for chain in empirical_chains:
484             all_terminals.extend(chain)
485
486         freq = Counter(all_terminals)
487         total = len(all_terminals)
488         return {sym: count/total for sym, count in freq.items
              ()}
489
490     def evaluate_meaningfulness(self):
```

```
491         """
492         Bewertet die Verst ndlichkeit der Grammatik.
493         """
494         print("\n" + "=" * 70)
495         print("VALIDIERUNG: VERST NDLICHKEIT (XAI-Kriterium
                1)")
496         print("=" * 70)
497 
498         # Pr fe , ob alle Nonterminale interpretierbare Namen
                haben
499         meaningful_count = 0
500         for nt in self.inducer.nonterminals:
501             if nt.startswith('NT_') and len(nt) > 3:
502                 meaningful_count += 1
503 
504         meaningful_ratio = meaningful_count / len(self.
                inducer.nonterminals) if self.inducer.nonterminals
                else 0
505 
506         print(f"\nNonterminale insgesamt: {len(self.inducer.
                nonterminals)}")
507         print(f"Davon interpretierbar benannt: {
                meaningful_count} ({meaningful_ratio:.1%})")
508 
509         # Dokumentierte Interpretationen
510         print(f"\nDokumentierte Interpretationsentscheidungen
                : {len(self.inducer.reflection.interpretation_log)
                }")
511 
512         # Beispiel-Interpretationen
513         if self.inducer.reflection.interpretation_log:
514             print("\nBeispiel-Interpretationen:")
515             for i, log in enumerate(self.inducer.reflection.
                    interpretation_log[:3]):
516                 seq_str = '     '.join([str(s) for s in log['
                        sequence']])
517                 print(f"  {i+1}. {seq_str}     {log['
                        new_nonterminal']}")
518                 print(f"     Begr ndung: {log['rationale']}"
                        )
```

```
519
520         return meaningful_ratio
521
522     def evaluate_accuracy(self, n_generated=500):
523         """
524         Bewertet die Genauigkeit der Grammatik.
525         """
526         print("\n" + "=" * 70)
527         print("VALIDIERUNG: GENAUIGKEIT (XAI-Kriterium 2)")
528         print("=" * 70)
529
530         generator = InterpretiveGenerator(
531             self.inducer.rules,
532             self.inducer.terminals,
533             self.inducer.start_symbol,
534             self.inducer.reflection
535         )
536
537         # Generiere viele Ketten
538         all_generated = []
539         for _ in range(n_generated):
540             chain, _ = generator.generate_with_interpretation
                ()
541             all_generated.extend(chain)
542
543         # Berechne generierte H ufigkeiten
544         gen_freq = Counter(all_generated)
545         total_gen = len(all_generated)
546         gen_dist = {sym: count/total_gen for sym, count in
            gen_freq.items() if total_gen > 0}
547
548         # Korrelationsberechnung f r gemeinsame Symbole
549         common_symbols = sorted(set(self.original_freq.keys()
            ) & set(gen_dist.keys()))
550         if common_symbols and len(common_symbols) > 1:
551             orig_values = [self.original_freq[sym] for sym in
                 common_symbols]
552             gen_values = [gen_dist[sym] for sym in
                common_symbols]
553
```

```
554             correlation, p_value = pearsonr(orig_values,
                    gen_values)
555
556             print(f"\nKorrelation (r): {correlation:.4f}")
557             print(f"Signifikanz (p): {p_value:.4f}")
558             print(f"Basis: {len(common_symbols)} gemeinsame
                    Symbole")
559
560             # Detaillierte Tabelle
561             print("\nVergleich der H ufigkeiten (Top 8):")
562             table_data = []
563             for sym in common_symbols[:8]:
564                 table_data.append([
565                     sym,
566                     f"{self.original_freq[sym]:.4f}",
567                     f"{gen_dist[sym]:.4f}",
568                     f"{abs(self.original_freq[sym] - gen_dist
                            [sym]):.4f}"
569                 ])
570
571             print(tabulate(table_data,
572                            headers=["Symbol", "Empirisch", "
                                   Generiert", "Differenz"],
573                            tablefmt="grid"))
574
575             return correlation, p_value
576         else:
577             print("Nicht gen gend gemeinsame Symbole f r
                    Korrelationsberechnung")
578             return 0, 1
579
580     def evaluate_knowledge_limits(self):
581         """
582         Dokumentiert die Wissensgrenzen der Grammatik.
583         """
584         print("\n" + "=" * 70)
585         print("VALIDIERUNG: WISSENSGRENZEN (XAI-Kriterium 3)"
                )
586         print("=" * 70)
587
```

```
588         print("\nDie Grammatik ist eine EXPLIKATION, keine
               Entdeckung:")
589         print("      Sie basiert auf 8 Transkripten von
               Verkaufsgespr chen")
590         print("      Die Terminalzeichen wurden durch
               qualitative Interpretation gewonnen")
591         print("      Die Nonterminale repr sentieren
               INTERPRETATIVE KATEGORIEN")
592
593         print("\nGRENZEN DER GRAMMATIK:")
594         print("      Keine Generalisierung  ber  den
               Datensatz hinaus")
595         print("      Keine Prognosef higkeit  f  r  neue
               Kontexte")
596         print("      Abh ngig von der initialen
               Kategorienbildung")
597         print("      Alternative Interpretationen sind
               m glich")
598
599         # Dokumentiere nicht abgedeckte Muster
600         observed_pairs = set()
601         for chain in empirical_chains:
602             for i in range(len(chain) - 1):
603                 observed_pairs.add((chain[i], chain[i+1]))
604
605         print(f"\nABGEDECKTE MUSTER:")
606         print(f"      Beobachtete    bergnge   : {len(
               observed_pairs)}")
607         print(f"      In Grammatik erfasste Nonterminale: {
               len(self.inducer.nonterminals)}")
608
609 #
    ================================================================================

610 # 6. HAUPTAUSF HRUNG
611 #
    ================================================================================

612
613 def main():
```

```
    """
    Hauptfunktion mit methodologischer Rahmung.
    """
    print("=" * 70)
    print("ALGORITHMISCH REKURSIVE SEQUENZANALYSE 3.0")
    print("HIERARCHISCHE GRAMMATIKINDUKTION")
    print("=" * 70)

    # 1. Grammatik induzieren
    inducer = GrammarInducer()
    compressed_chains = inducer.induce_grammar(
        empirical_chains)

    # 2. Methodologische Reflexion
    inducer.reflection.print_methodological_summary()

    # 3. Induzierte Grammatik anzeigen
    print("\n" + "=" * 70)
    print("INDUZIERTE GRAMMATIK")
    print("=" * 70)
    print(f"\nTerminale ({len(inducer.terminals)}): {sorted(
        inducer.terminals)}")
    print(f"Nonterminale ({len(inducer.nonterminals)}): {
        sorted(inducer.nonterminals)}")
    if inducer.start_symbol:
        print(f"Startsymbol: {inducer.start_symbol}")

    print("\nPRODUKTIONSREGELN (mit Wahrscheinlichkeiten):")
    for nonterminal in sorted(inducer.rules.keys()):
        productions = inducer.rules[nonterminal]
        if productions:
            prod_strings = []
            for prod, prob in productions:
                # Stelle sicher, dass prod eine Liste ist
                if isinstance(prod, tuple):
                    prod = list(prod)
                prod_str = '     '.join([str(s) for s in prod
                    ])
                # Stelle sicher, dass prob ein Float ist
```

```
649                 prob_float = float(prob) if not isinstance(
                        prob, (int, float)) else prob
650                 prod_strings.append(f"{prod_str} [{prob_float
                        :.3f}]")
651             print(f"\n{nonterminal}     {' | '.join(
                    prod_strings)}")
652
653     # 4. Beispiele mit Interpretation generieren
654     print("\n" + "=" * 70)
655     print("BEISPIELE MIT INTERPRETATION")
656     print("=" * 70)
657
658     generator = InterpretiveGenerator(
659         inducer.rules,
660         inducer.terminals,
661         inducer.start_symbol,
662         inducer.reflection
663     )
664
665     for i in range(3):
666         chain, interpretation = generator.
                generate_with_interpretation()
667         print(f"\nBeispiel {i+1}:")
668         chain_str = '     '.join([str(s) for s in chain
                [:10]])
669         print(f"  Kette: {chain_str}" + ("..." if len(chain)
                > 10 else ""))
670         print("  Interpretation:")
671         for j, step in enumerate(interpretation[:5]):
672             print(f"    {j+1}. {step}")
673         if len(interpretation) > 5:
674             print("    ...")
675
676     # 5. XAI-Validierung
677     validator = XAIValidator(inducer)
678     validator.evaluate_meaningfulness()
679     validator.evaluate_accuracy(n_generated=500)
680     validator.evaluate_knowledge_limits()
681
682     # 6. Grammatik exportieren
```

```
683     print("\n" + "=" * 70)
684     print("EXPORT DER GRAMMATIK")
685     print("=" * 70)
686 
687     with open("induzierte_grammatik_mit_interpretation.txt",
            'w', encoding='utf-8') as f:
688         f.write("# INDUZIERTE PCFG MIT INTERPRETATION\n")
689         f.write("# ================================\n\n")
690         f.write(f"## DATENGRUNDLAGE\n")
691         f.write(f"{len(empirical_chains)} Transkripte von
                Verkaufsgespr chen\n\n")
692 
693         f.write("## TERMINALE (qualitative Kategorien)\n")
694         for sym in sorted(inducer.terminals):
695             f.write(f"{sym}: {inducer.reflection.
                    _interpretiere_symbol(sym)}\n")
696 
697         f.write("\n## NONTERMINALE (interpretative Kategorien
                )\n")
698         for log in inducer.reflection.interpretation_log:
699             seq_str = '     '.join([str(s) for s in log['
                    sequence']])
700             f.write(f"\n{log['new_nonterminal']}\n")
701             f.write(f"  Muster: {seq_str}\n")
702             mapping = inducer.reflection.
                    sequence_meaning_mapping.get(tuple(log['
                    sequence']), {})
703             if mapping:
704                 f.write(f"  Bedeutung: {mapping.get('
                        bedeutung', '')}\n")
705             f.write(f"  Begr ndung: {log['rationale']}\n")
706 
707         f.write("\n## PRODUKTIONSREGELN\n")
708         for nt in sorted(inducer.rules.keys()):
709             prods = inducer.rules[nt]
710             for prod, prob in prods:
711                 if isinstance(prod, tuple):
712                     prod = list(prod)
713                 prod_str = ' '.join([str(s) for s in prod])
```

```
                    prob_float = float(prob) if not isinstance(
                        prob, (int, float)) else prob
                    f.write(f"{nt}     {prod_str} [{prob_float:.3
                        f}]\n")

    print(f"\nGrammatik exportiert als '
        induzierte_grammatik_mit_interpretation.txt'")

    print("\n" + "=" * 70)
    print("ALGORITHMISCH REKURSIVE SEQUENZANALYSE
        ABGESCHLOSSEN")
    print("=" * 70)

if __name__ == "__main__":
    main()
```

Listing 2: Algorithmisch Rekursive Sequenzanalyse 3.0 - Hierarchische Grammatikinduktion