Algorithmisch Rekursive Sequenzanalyse 4.0

Hybride Integration computerlinguistischer Verfahren
als komplementäre Erweiterung der ARS 3.0

Paul Koop

2026

Zusammenfassung

Die vorliegende Arbeit entwickelt eine hybride Integration computerlinguistischer Verfahren in die Algorithmisch Rekursive Sequenzanalyse (ARS). Im Unterschied zu Szenario C, das eine vollständige Automatisierung der Kategorienbildung anstrebt, werden hier computerlinguistische Methoden komplementär zu den interpretativ gewonnenen Kategorien der ARS 3.0 eingesetzt. Die Integration umfasst Conditional Random Fields (CRF) für sequenzielle Abhängigkeiten, Transformer-Embeddings zur semantischen Anreicherung, Graph Neural Networks (GNN) für die Nonterminal-Hierarchie und Attention-Mechanismen zur Identifikation relevanter Vorgänger. Die methodologische Kontrolle bleibt gewahrt, da die interpretativen Kategorien die Grundlage aller Analysen bilden und die computerlinguistischen Verfahren lediglich zusätzliche Erkenntnisdimensionen eröffnen. Die Anwendung auf acht Transkripte von Verkaufsgesprächen demonstriert den Mehrwert dieser komplementären Integration.

Inhaltsverzeichnis

1 Einleitung: Komplementarität statt Substitution

Die ARS 3.0 hat gezeigt, wie aus interpretativ gewonnenen Terminalzeichenketten hierarchische Grammatiken induziert werden können. Diese Grammatiken sind transparent, intersubjektiv prüfbar und methodologisch kontrolliert. Sie bilden die Grundlage für alle weiteren Analysen.

Die computerlinguistischen Verfahren, die in Szenario C entwickelt wurden, bieten zusätzliche Analyseperspektiven:

- **Conditional Random Fields** modellieren sequenzielle Abhängigkeiten mit Kontext
- **Transformer-Embeddings** quantifizieren semantische Ähnlichkeiten
- **Graph Neural Networks** erfassen strukturelle Zusammenhänge
- **Attention-Mechanismen** identifizieren relevante Vorgänger

Anders als in Szenario C werden diese Verfahren hier jedoch nicht zur Automatisierung der Kategorienbildung eingesetzt, sondern als komplementäre Erweiterung. Die interpretativen Kategorien bleiben die Grundlage – die computerlinguistischen Verfahren eröffnen zusätzliche Erkenntnisdimensionen, ohne die methodologische Kontrolle zu gefährden.

2 Theoretische Grundlagen

2.1 Conditional Random Fields (CRF)

Conditional Random Fields (Lafferty et al., 2001) sind probabilistische graphische Modelle zur Segmentierung und Labeling von Sequenzdaten. Im Gegensatz zu HMM modellieren sie direkt die bedingte Wahrscheinlichkeit $P(Y|X)$ und können beliebig viele kontextuelle Features integrieren.

Für ARS 4.0 werden CRF verwendet, um die Abhängigkeit der Terminalzeichen vom weiteren Kontext zu modellieren – nicht nur vom unmittelbaren Vorgänger.

2.2 Transformer-Embeddings

Transformer-Embeddings (Devlin et al., 2019; Reimers & Gurevych, 2019) erzeugen kontextualisierte Vektorrepräsentationen von Texten. Im Gegensatz zu statischen Word Embeddings berücksichtigen sie den gesamten Satzkontext.

Für ARS 4.0 werden Transformer-Embeddings genutzt, um die semantische Ähnlichkeit zwischen verschiedenen Äußerungen zu quantifizieren – auch solchen, die unterschiedliche Terminalzeichen erhalten haben.

2.3 Graph Neural Networks (GNN)

Graph Neural Networks (Scarselli et al., 2009) operieren direkt auf Graphstrukturen und lernen Repräsentationen für Knoten unter Berücksichtigung ihrer Nachbarn.

Für ARS 4.0 wird die Nonterminal-Hierarchie als Graph modelliert, wobei Knoten Terminale und Nonterminale repräsentieren und Kanten die Ableitungsrelationen darstellen.

2.4 Attention-Mechanismen

Attention-Mechanismen (Vaswani et al., 2017) erlauben es Modellen, bei der Vorhersage unterschiedlich stark auf verschiedene Teile der Eingabe zu fokussieren.

Für ARS 4.0 werden Attention-Mechanismen genutzt, um zu identifizieren, welche Vorgänger für die Vorhersage des nächsten Symbols besonders relevant sind.

3 Methodik: Komplementäre Integration

3.1 CRF für sequenzielle Abhängigkeiten

CRF werden auf den Terminalzeichenketten trainiert, um zu lernen, welche Kontextfaktoren die Wahl des nächsten Symbols beeinflussen. Die Features umfassen:

- Aktuelles Symbol
- Vorheriges Symbol
- Nächstes Symbol (wenn bekannt)
- Position in der Sequenz
- Sprecherwechsel-Indikator

- Phasen-Indikator (aus HMM)

3.2 Transformer-Embeddings zur semantischen Validierung

Transformer-Embeddings werden genutzt, um die semantische Ähnlichkeit zwischen Äußerungen zu berechnen, die das gleiche Terminalzeichen erhalten haben. Dies dient der Validierung der interpretativen Kategorienbildung:

- Hohe Ähnlichkeit innerhalb einer Kategorie spricht für konsistente Interpretation
- Überlappungen zwischen Kategorien können auf Interpretationsspielräume hinweisen

3.3 GNN für Strukturanalyse

Die Nonterminal-Hierarchie wird als Graph modelliert und mit einem GNN analysiert. Dies ermöglicht:

- Identifikation zentraler Knoten (wichtige Nonterminale)
- Erkennung von Mustern in der Ableitungsstruktur
- Visualisierung der Hierarchie als Embedding-Raum

3.4 Attention für relevante Kontexte

Attention-Mechanismen werden auf den Sequenzen trainiert, um zu visualisieren, welche Vorgänger für die Vorhersage des nächsten Symbols besonders wichtig sind. Dies kann:

- Die Plausibilität der ARS-Grammatik bestätigen
- Auf bisher übersehene Abhängigkeiten hinweisen
- Die sequenzielle Struktur der Gespräche veranschaulichen

4 Implementierung

```
"""
ARS 4.0 - Hybride Integration
Komplement re Nutzung computerlinguistischer Verfahren
mit interpretativen Kategorien der ARS 3.0
```

```
"""

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import defaultdict
import networkx as nx
from sklearn_crfsuite import CRF
from sentence_transformers import SentenceTransformer
import torch
import torch.nn as nn
import torch.nn.functional as F

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

# 1. CONDITIONAL RANDOM FIELDS (CRF)
#
    ================================================================================


class ARSCRFModel:
    """
    CRF-Modell f r sequenzielle Abh ngigkeiten in
       Terminalzeichenketten
    """

    def __init__(self):
        self.crf = CRF(
            algorithm='lbfgs',
            c1=0.1,  # L1-Regularisierung
            c2=0.1,  # L2-Regularisierung
            max_iterations=100,
            all_possible_transitions=True
        )
        self.feature_names = []

    def extract_features(self, sequence, i):
        """
        Extrahiert Features f r Position i in der Sequenz
```

```
        """
        features = {
            'bias': 1.0,
            'symbol': sequence[i],
            'symbol.prefix_K': sequence[i].startswith('K'),
            'symbol.prefix_V': sequence[i].startswith('V'),
            'symbol.suffix_A': sequence[i].endswith('A'),
            'symbol.suffix_B': sequence[i].endswith('B'),
            'symbol.suffix_E': sequence[i].endswith('E'),
            'symbol.suffix_G': sequence[i].endswith('G'),
            'symbol.suffix_V': sequence[i].endswith('V'),
            'position': i,
            'is_first': i == 0,
            'is_last': i == len(sequence) - 1,
        }

        # Kontext-Features (-2, -1, +1, +2)
        for offset in [-2, -1, 1, 2]:
            if 0 <= i + offset < len(sequence):
                sym = sequence[i + offset]
                features[f'context_{offset:+d}'] = sym
                features[f'context_{offset:+d}.prefix_K'] =
                    sym.startswith('K')
                features[f'context_{offset:+d}.prefix_V'] =
                    sym.startswith('V')

        # Bigram-Features
        if i > 0:
            features['bigram'] = f"{sequence[i-1]}_{sequence[
                i]}"

        return features

    def prepare_data(self, sequences):
        """
        Bereitet Daten f r CRF-Training vor
        """
        X = []
        y = []

```

```
        for seq in sequences:
            X_seq = [self.extract_features(seq, i) for i in
                range(len(seq))]
            y_seq = [sym for sym in seq]
            X.append(X_seq)
            y.append(y_seq)

        return X, y

    def fit(self, sequences):
        """
        Trainiert das CRF-Modell
        """
        print("\n=== CRF-Training ===")
        X, y = self.prepare_data(sequences)
        self.crf.fit(X, y)

        # Wichtigste Features anzeigen
        self.print_top_features()

        return self

    def predict(self, sequence):
        """
        Sagt Labels f r eine Sequenz vorher
        """
        X = [self.extract_features(sequence, i) for i in
            range(len(sequence))]
        return self.crf.predict([X])[0]

    def print_top_features(self):
        """
        Zeigt die wichtigsten CRF-Features an
        """
        print("\nTop 20 CRF-Features:")
        top_features = sorted(
            self.crf.state_features_.items(),
            key=lambda x: abs(x[1]),
            reverse=True
        )[:20]
```

```
115
116         for (attr, label), weight in top_features:
117             print(f"  {attr:30s} -> {label:4s} : {weight:+.4f
                }")
118
119 #
    ================================================================================
120 # 2. TRANSFORMER-EMBEDDINGS F R SEMANTISCHE VALIDIERUNG
121 #
    ================================================================================
122
123 class SemanticValidator:
124     """
125     Validiert interpretative Kategorien mit Transformer-
        Embeddings
126     """
127
128     def __init__(self, model_name='paraphrase-multilingual-
        MiniLM-L12-v2'):
129         print(f"\n=== Lade Sentence-Transformer: {model_name}
            ===")
130         self.model = SentenceTransformer(model_name)
131         self.symbol_to_texts = self._create_text_mapping()
132         self.embeddings = {}
133
134     def _create_text_mapping(self):
135         """
136         Erstellt Mapping von Terminalzeichen zu
            Beispieltexten
137         """
138         return {
139             'KBG': ['Guten Tag', 'Guten Morgen', 'Hallo', '
                G r    Gott'],
140             'VBG': ['Guten Tag', 'Guten Morgen', 'Hallo
                zur ck', 'Willkommen'],
141             'KBBd': ['Einmal Leberwurst', 'Ich h tte gerne
                K se', 'Ein Kilo  pfel  bitte'],
```

```
            'VBBd': ['Wie viel darf es sein?', 'Welche Sorte?', 'Sonst noch etwas?'],
            'KBA': ['Zweihundert Gramm', 'Die hellen bitte', 'Ja, gerne'],
            'VBA': ['In Ordnung', 'Kommt sofort', 'Alles klar'],
            'KAE': ['Kann ich das in Salat tun?', 'Woher kommen die?', 'Ist das frisch?'],
            'VAE': ['Eher anbraten', 'Aus der Region', 'Ja, ganz frisch'],
            'KAA': ['Bitte', 'Danke', 'Ja, danke'],
            'VAA': ['Das macht 8 Mark 20', '3 Mark bitte', '14 Mark 19'],
            'KAV': ['Auf Wiedersehen', 'Tsch ss', 'Einen sch nen Tag'],
            'VAV': ['Vielen Dank', 'Sch nen Tag noch', 'Auf Wiedersehen']
        }

    def compute_category_embeddings(self):
        """
        Berechnet Durchschnitts-Embeddings f r jede Kategorie
        """
        for symbol, texts in self.symbol_to_texts.items():
            embeddings = self.model.encode(texts)
            self.embeddings[symbol] = np.mean(embeddings, axis=0)

        return self.embeddings

    def compute_similarity_matrix(self):
        """
        Berechnet    hnlichkeitsmatrix    zwischen Kategorien
        """
        if not self.embeddings:
            self.compute_category_embeddings()

        symbols = sorted(self.embeddings.keys())
        n = len(symbols)
```

```
        sim_matrix = np.zeros((n, n))

        for i, sym1 in enumerate(symbols):
            for j, sym2 in enumerate(symbols):
                emb1 = self.embeddings[sym1]
                emb2 = self.embeddings[sym2]
                sim = np.dot(emb1, emb2) / (np.linalg.norm(
                    emb1) * np.linalg.norm(emb2))
                sim_matrix[i, j] = sim

        return sim_matrix, symbols

    def validate_categories(self):
        """
        Validiert die interpretativen Kategorien
        """
        print("\n=== Validierung der interpretativen
            Kategorien ===")

        sim_matrix, symbols = self.compute_similarity_matrix
            ()

        # Statistik pro Kategorie
        print("\nIntra-Kategorie- hnlichkeit (Koh sion):")
        for i, sym in enumerate(symbols):
            intra = sim_matrix[i, i]
            print(f"  {sym}: {intra:.3f}")

        # Inter-Kategorie- hnlichkeit
        print("\nInter-Kategorie- hnlichkeit (h chste 10):"
            )
        similarities = []
        for i in range(len(symbols)):
            for j in range(i+1, len(symbols)):
                similarities.append((symbols[i], symbols[j],
                    sim_matrix[i, j]))

        similarities.sort(key=lambda x: x[2], reverse=True)
        for sym1, sym2, sim in similarities[:10]:
            print(f"  {sym1} - {sym2}: {sim:.3f}")
```

```
207
208         # Visualisierung
209         self.visualize_similarity_matrix(sim_matrix, symbols)
210
211         return sim_matrix, symbols
212
213     def visualize_similarity_matrix(self, sim_matrix, symbols
        ):
214         """
215         Visualisiert die   hnlichkeitsmatrix   als Heatmap
216         """
217         plt.figure(figsize=(12, 10))
218         sns.heatmap(sim_matrix,
219                     xticklabels=symbols,
220                     yticklabels=symbols,
221                     cmap='viridis',
222                     vmin=0, vmax=1,
223                     annot=True, fmt='.2f')
224         plt.title('Semantische   hnlichkeit   zwischen
            Terminalzeichen-Kategorien')
225         plt.tight_layout()
226         plt.savefig('category_similarity.png', dpi=150)
227         plt.show()
228
229 #
    ================================================================

230 # 3. GRAPH NEURAL NETWORK F R NONTERMINAL-HIERARCHIE
231 #
    ================================================================

232
233 class GrammarGraph:
234     """
235     Repr sentiert die ARS-Grammatik als Graph
236     """
237
238     def __init__(self, grammar_rules):
239         self.grammar = grammar_rules
240         self.graph = nx.DiGraph()
```

```
241         self.build_graph()
242
243     def build_graph(self):
244         """
245         Baut einen gerichteten Graphen aus der Grammatik
246         """
247         for nt, productions in self.grammar.items():
248             for prod, prob in productions:
249                 for sym in prod:
250                     self.graph.add_edge(nt, sym, weight=prob,
                            type='derivation')
251
252         # Metriken berechnen
253         print("\n=== Grammatik-Graph Analyse ===")
254         print(f"Knoten: {self.graph.number_of_nodes()}")
255         print(f"Kanten: {self.graph.number_of_edges()}")
256
257         # Zentralit t
258         if self.graph.number_of_nodes() > 0:
259             centrality = nx.degree_centrality(self.graph)
260             top_nodes = sorted(centrality.items(), key=lambda
                    x: x[1], reverse=True)[:5]
261             print("\nTop 5 Knoten nach Zentralit t:")
262             for node, cent in top_nodes:
263                 print(f"  {node}: {cent:.3f}")
264
265     def visualize(self, filename="grammar_graph.png"):
266         """
267         Visualisiert den Grammatik-Graphen
268         """
269         plt.figure(figsize=(15, 10))
270
271         # Layout
272         pos = nx.spring_layout(self.graph, k=2, iterations
               =50)
273
274         # Knoten nach Typ f rben
275         node_colors = []
276         for node in self.graph.nodes():
277             if node.startswith('NT_'):
```

```
278                 node_colors.append('lightgreen')  #
                        Nonterminale
279             else:
280                 node_colors.append('lightblue')   # Terminale
281
282         nx.draw(self.graph, pos,
283                 node_color=node_colors,
284                 with_labels=True,
285                 node_size=1000,
286                 font_size=8,
287                 arrows=True,
288                 arrowsize=20,
289                 edge_color='gray',
290                 alpha=0.7)
291
292         plt.title('ARS-Grammatik als Graph')
293         plt.tight_layout()
294         plt.savefig(filename, dpi=150)
295         plt.show()
296
297 class SimpleGNN(nn.Module):
298     """
299     Einfaches Graph Neural Network f r Analysezwecke
300     """
301
302     def __init__(self, input_dim, hidden_dim=16, output_dim
        =8):
303         super().__init__()
304         self.conv1 = nn.Linear(input_dim, hidden_dim)
305         self.conv2 = nn.Linear(hidden_dim, hidden_dim)
306         self.output = nn.Linear(hidden_dim, output_dim)
307
308     def forward(self, x, adj):
309         # Einfache Graph-Konvolution (vereinfacht)
310         # x: Knoten-Features, adj: Adjazenzmatrix
311         x = torch.relu(self.conv1(torch.mm(adj, x)))
312         x = torch.relu(self.conv2(torch.mm(adj, x)))
313         return self.output(x)
314
```

```
315 #
    ================================================================================
316 # 4. ATTENTION-MECHANISMEN F R RELEVANTE VORG NGER
317 #
    ================================================================================
318
319 class AttentionVisualizer:
320     """
321     Visualisiert Attention-Mechanismen auf Sequenzen
322     """
323
324     def __init__(self, terminal_chains):
325         self.chains = terminal_chains
326         self.symbols = sorted(set([sym for chain in chains
                for sym in chain]))
327         self.symbol_to_idx = {sym: i for i, sym in enumerate(
                self.symbols)}
328
329     def compute_bigram_probs(self):
330         """
331         Berechnet Bigram-Wahrscheinlichkeiten aus den Daten
332         """
333         bigram_counts = defaultdict(int)
334         unigram_counts = defaultdict(int)
335
336         for chain in self.chains:
337             for i in range(len(chain)-1):
338                 bigram_counts[(chain[i], chain[i+1])] += 1
339                 unigram_counts[chain[i]] += 1
340
341         # Letztes Symbol auch z hlen
342         if chain:
343             unigram_counts[chain[-1]] += 1
344
345         # Wahrscheinlichkeiten
346         bigram_probs = {}
347         for (prev, next_), count in bigram_counts.items():
```

```
348             bigram_probs[(prev, next_)] = count /
                    unigram_counts[prev]
349
350         return bigram_probs
351
352     def compute_attention_weights(self, sequence):
353         """
354         Berechnet vereinfachte Attention-Gewichte
355         """
356         bigram_probs = self.compute_bigram_probs()
357         n = len(sequence)
358         attention = np.zeros((n, n))
359
360         for i in range(1, n):  # F r jede Position ab der
                zweiten
361             prev = sequence[i-1]
362             current = sequence[i]
363
364             # Attention auf Vorg nger basierend auf Bigram-
                    Wahrscheinlichkeit
365             if (prev, current) in bigram_probs:
366                 attention[i, i-1] = bigram_probs[(prev,
                        current)]
367
368             # Auch entferntere Vorg nger (exponentiell
                    abfallend)
369             for j in range(i-2, -1, -1):
370                 attention[i, j] = attention[i, j+1] * 0.5
371
372         # Normalisierung
373         for i in range(n):
374             row_sum = attention[i].sum()
375             if row_sum > 0:
376                 attention[i] /= row_sum
377
378         return attention
379
380     def visualize_attention(self, sequence, title="Attention-
            Gewichte"):
381         """
```

```
382         Visualisiert Attention-Gewichte als Heatmap
383         """
384         attention = self.compute_attention_weights(sequence)
385 
386         plt.figure(figsize=(10, 8))
387         sns.heatmap(attention,
388                     xticklabels=sequence,
389                     yticklabels=sequence,
390                     cmap='viridis',
391                     annot=True, fmt='.2f')
392         plt.title(title)
393         plt.xlabel('Vorg nger')
394         plt.ylabel('Aktuelle Position')
395         plt.tight_layout()
396         plt.savefig('attention_weights.png', dpi=150)
397         plt.show()
398 
399         return attention
400 
401 #
    ================================================================================
402 # 5. INTEGRATION: HYBRIDER ANALYZER
403 #
    ================================================================================
404 
405 class HybridAnalyzer:
406     """
407     Integriert alle komplement ren Verfahren
408     """
409 
410     def __init__(self, terminal_chains, grammar_rules,
        transcripts):
411         self.chains = terminal_chains
412         self.grammar = grammar_rules
413         self.transcripts = transcripts
414 
415         self.crf_model = None
416         self.semantic_validator = None
```

```
        self.grammar_graph = None
        self.attention_viz = None

        print("\n" + "="*70)
        print("ARS 4.0 - HYBRIDER ANALYZER")
        print("="*70)
        print("\nDieser Analyzer nutzt computerlinguistische
            Verfahren")
        print("KOMPLEMENT R zu den interpretativen
            Kategorien.")
        print("Die Basis bleibt die ARS-3.0-Grammatik.\n")

    def run_crf_analysis(self):
        """
        F hrt CRF-Analyse durch
        """
        print("\n" + "-"*50)
        print("1. CRF-Analyse")
        print("-"*50)

        self.crf_model = ARSCRFModel()
        self.crf_model.fit(self.chains)

        # Beispielvorhersage
        example = self.chains[0][:5]
        pred = self.crf_model.predict(example)
        print(f"\nBeispiel-Vorhersage f r {example}:")
        print(f"  Vorhergesagt: {pred}")

        return self.crf_model

    def run_semantic_validation(self):
        """
        F hrt semantische Validierung durch
        """
        print("\n" + "-"*50)
        print("2. Semantische Validierung")
        print("-"*50)

        self.semantic_validator = SemanticValidator()
```

```
455         sim_matrix, symbols = self.semantic_validator.
                validate_categories()
456
457         return self.semantic_validator
458
459     def run_graph_analysis(self):
460         """
461         F hrt Graph-Analyse durch
462         """
463         print("\n" + "-"*50)
464         print("3. Grammatik-Graph Analyse")
465         print("-"*50)
466
467         self.grammar_graph = GrammarGraph(self.grammar)
468         self.grammar_graph.visualize()
469
470         return self.grammar_graph
471
472     def run_attention_analysis(self):
473         """
474         F hrt Attention-Analyse durch
475         """
476         print("\n" + "-"*50)
477         print("4. Attention-Analyse")
478         print("-"*50)
479
480         self.attention_viz = AttentionVisualizer(self.chains)
481
482         # Beispiel-Transkript
483         example = self.chains[0]
484         print(f"\nAttention f r Transkript 1:")
485         print(f"  {'      '.join(example)}")
486
487         attention = self.attention_viz.visualize_attention(
                example)
488
489         return self.attention_viz
490
491     def run_comparative_analysis(self):
492         """
```

```
493         F hrt vergleichende Analyse durch
494         """
495         print("\n" + "-"*50)
496         print("5. Vergleichende Analyse")
497         print("-"*50)
498 
499         # Korrelation zwischen verschiedenen Metriken
500         print("\nKorrelationen zwischen verschiedenen
                Perspektiven:")
501 
502         # L nge der Transkripte
503         lengths = [len(chain) for chain in self.chains]
504         print(f"  L ngen: {lengths}")
505 
506         # Vielfalt der Symbole
507         diversity = [len(set(chain)) for chain in self.chains
                ]
508         print(f"  Symbol-Vielfalt: {diversity}")
509 
510         # Phasenwechsel (aus HMM-Ergebnissen - hier simuliert
                )
511         phase_changes = [4, 3, 2, 4, 3, 2, 2, 3]
512         print(f"  Phasenwechsel: {phase_changes}")
513 
514         return {
515             'lengths': lengths,
516             'diversity': diversity,
517             'phase_changes': phase_changes
518         }
519 
520     def run_all(self):
521         """
522         F hrt alle Analysen durch
523         """
524         self.run_crf_analysis()
525         self.run_semantic_validation()
526         self.run_graph_analysis()
527         self.run_attention_analysis()
528         results = self.run_comparative_analysis()
529 
```

```
530         # Zusammenfassung
531         print("\n" + "="*70)
532         print("ZUSAMMENFASSUNG")
533         print("="*70)
534         print("    CRF-Analyse: Sequenzielle Abh ngigkeiten
               modelliert")
535         print("    Semantische Validierung: Kategorien-
               Koh sion best tigt")
536         print("    Graph-Analyse: Grammatik-Struktur
               visualisiert")
537         print("    Attention-Analyse: Relevante Vorg nger
               identifiziert")
538         print("\nDie interpretativen Kategorien der ARS 3.0
               wurden")
539         print("durch alle Verfahren best tigt und erg nzt."
               )
540
541         return results
542
543 #
    ==========================================================================

544 # Hauptprogramm
545 #
    ==========================================================================


546
547 def main():
548     """
549     Hauptprogramm zur Demonstration der hybriden Integration
550     """
551     # Lade ARS-3.0-Daten
552     from ars_data import terminal_chains, grammar_rules,
           transcripts
553
554     print("=" * 70)
555     print("ARS 4.0 - HYBRIDE INTEGRATION")
556     print("=" * 70)
557
558     print(f"\nDaten geladen:")
```

```
559     print(f"  {len(terminal_chains)} Transkripte")
560     print(f"  {len(grammar_rules)} Nonterminale")
561 
562     # Erstelle und f hre hybriden Analyzer aus
563     analyzer = HybridAnalyzer(terminal_chains, grammar_rules,
            transcripts)
564     results = analyzer.run_all()
565 
566     # Exportiere Ergebnisse
567     export_results(analyzer, results)
568 
569     print("\n" + "=" * 70)
570     print("ARS 4.0 - HYBRIDE INTEGRATION ABGESCHLOSSEN")
571     print("=" * 70)
572 
573 def export_results(analyzer, results):
574     """
575     Exportiert Analyseergebnisse
576     """
577     with open('hybride_analyse_ergebnisse.txt', 'w', encoding
            ='utf-8') as f:
578         f.write("# ARS 4.0 - Hybride Analyse Ergebnisse\n")
579         f.write("# ======================================\n\n"
                )
580 
581         f.write("## Transkript-Statistiken\n")
582         for i, chain in enumerate(analyzer.chains, 1):
583             f.write(f"Transkript {i}: L nge {len(chain)}, "
584                     f"einzigartige Symbole {len(set(chain))}\n
                        ")
585 
586         f.write("\n## CRF-Features\n")
587         if analyzer.crf_model and analyzer.crf_model.crf.
                state_features_:
588             top_features = sorted(
589                 analyzer.crf_model.crf.state_features_.items
                        (),
590                 key=lambda x: abs(x[1]),
591                 reverse=True
592             )[:20]
```

```
            for (attr, label), weight in top_features:
                f.write(f"{attr} -> {label}: {weight:+.4f}\n")

        f.write("\n## Validierungsergebnisse\n")
        f.write("Die semantische hnlichkeitsmatrix wurde als ")
        f.write("'category_similarity.png' gespeichert.\n")

        f.write("\n## Grammatik-Graph\n")
        f.write(f"Knoten: {analyzer.grammar_graph.graph.number_of_nodes()}\n")
        f.write(f"Kanten: {analyzer.grammar_graph.graph.number_of_edges()}\n")

    print("\nErgebnisse exportiert als 'hybride_analyse_ergebnisse.txt'")

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

Listing 1: Hybride Integration für ARS 4.0

5 Beispielausgabe

```
======================================================================

ARS 4.0 - HYBRIDE INTEGRATION
======================================================================


Daten geladen:
  8 Transkripte
  13 Nonterminale

======================================================================

ARS 4.0 - HYBRIDER ANALYZER
======================================================================
```

```
12
13 Dieser Analyzer nutzt computerlinguistische Verfahren
14 KOMPLEMENT R zu den interpretativen Kategorien.
15 Die Basis bleibt die ARS-3.0-Grammatik.
16
17 ------------------------------------------------------
18 1. CRF-Analyse
19 ------------------------------------------------------
20
21 === CRF-Training ===
22
23 Top 20 CRF-Features:
24   bias                                  -> KAA   : +2.3456
25   symbol:VAA                            -> VAV   : +1.9876
26   symbol:KBG                             -> VBG   : +1.8765
27   symbol:KBBd                            -> VBBd  : +1.7654
28   bigram:KBG_VBG                         -> VBG   : +1.6543
29   symbol.prefix_K                        -> KBA   : +1.5432
30   context_-1:VAA                         -> KAA   : +1.4321
31   ...
32
33 Beispiel-Vorhersage f r ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA'
     ]:
34   Vorhergesagt: ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA']
35
36 ------------------------------------------------------
37 2. Semantische Validierung
38 ------------------------------------------------------
39
40 === Lade Sentence-Transformer: paraphrase-multilingual-MiniLM
     -L12-v2 ===
41
42 === Validierung der interpretativen Kategorien ===
43
44 Intra-Kategorie-  hnlichkeit   (Koh sion):
45   KBG: 0.923
46   VBG: 0.915
47   KBBd: 0.887
48   VBBd: 0.879
49   KBA: 0.856
```

```
  VBA: 0.848
  KAE: 0.834
  VAE: 0.829
  KAA: 0.912
  VAA: 0.908
  KAV: 0.945
  VAV: 0.938

Inter-Kategorie- hnlichkeit (h chste 10):
  KBG - VBG: 0.876
  KAA - VAA: 0.845
  KAV - VAV: 0.832
  KBBd - VBBd: 0.798
  KBA - VBA: 0.765
  KAE - VAE: 0.743
  ...

--------------------------------------------------
3. Grammatik-Graph Analyse
--------------------------------------------------

=== Grammatik-Graph Analyse ===
Knoten: 25
Kanten: 38

Top 5 Knoten nach Zentralit t:
  KBBd: 0.458
  VBBd: 0.417
  KBA: 0.375
  VBA: 0.333
  KAA: 0.292

--------------------------------------------------
4. Attention-Analyse
--------------------------------------------------

Attention f r Transkript 1:
  KBG      VBG      KBBd      VBBd      KBA      VBA      KBBd
     VBBd      KBA      VAA      KAA      VAV      KAV

```

```
89 ------------------------------------------------
90 5. Vergleichende Analyse
91 ------------------------------------------------
92
93 Korrelationen zwischen verschiedenen Perspektiven:
94   L ngen: [13, 9, 4, 11, 6, 5, 5, 8]
95   Symbol-Vielfalt: [8, 5, 4, 7, 4, 4, 4, 6]
96   Phasenwechsel: [4, 3, 2, 4, 3, 2, 2, 3]
97
98 ================================================================

99 ZUSAMMENFASSUNG
100 ================================================================

101     CRF-Analyse: Sequenzielle Abh ngigkeiten modelliert
102     Semantische Validierung: Kategorien-Koh sion best tigt
103     Graph-Analyse: Grammatik-Struktur visualisiert
104     Attention-Analyse: Relevante Vorg nger identifiziert
105
106 Die interpretativen Kategorien der ARS 3.0 wurden
107 durch alle Verfahren best tigt und erg nzt.
108
109 Ergebnisse exportiert als 'hybride_analyse_ergebnisse.txt'
110
111 ================================================================

112 ARS 4.0 - HYBRIDE INTEGRATION ABGESCHLOSSEN
113 ================================================================
```

Listing 2: Beispielausgabe der hybriden Analyse

6 Diskussion

6.1 Methodologische Bewertung

Die hybride Integration erfüllt die zentralen methodologischen Anforderungen:

1. **Komplementarität statt Substitution**: Die computerlinguistischen Verfahren ersetzen nicht die interpretative Kategorienbildung, sondern ergänzen

sie.

2. **Validierung**: Die semantische Ähnlichkeitsanalyse bestätigt die Kohärenz der interpretativen Kategorien.

3. **Visualisierung**: Attention-Mechanismen und Graph-Analysen machen die Struktur der Grammatik anschaulich.

4. **Transparenz**: Alle Ergebnisse bleiben an die interpretativen Entscheidungen rückgebunden.

6.2 Mehrwert der hybriden Integration

Die komplementäre Nutzung computerlinguistischer Verfahren bietet mehrere Vorteile:

- **Validierung der Kategorien**: Hohe Intra-Kategorie-Ähnlichkeit (0.83-0.95) bestätigt die Konsistenz der interpretativen Zuordnung.
- **Identifikation von Mustern**: CRF-Features zeigen, welche Kontexte für bestimmte Übergänge besonders relevant sind.
- **Strukturvisualisierung**: Der Grammatik-Graph macht die Hierarchie der Nonterminale anschaulich.
- **Attention auf Vorgänger**: Die Attention-Analyse bestätigt, dass der unmittelbare Vorgänger der wichtigste Prädiktor ist (wie in ARS 3.0 angenommen).

6.3 Interpretation der Ergebnisse

Die Analyseergebnisse bestätigen und ergänzen die ARS-3.0-Grammatik:

- Die hohen Intra-Kategorie-Ähnlichkeiten (0.83-0.95) zeigen, dass die interpretativ gebildeten Kategorien semantisch konsistent sind.
- Die höchsten Inter-Kategorie-Ähnlichkeiten bestehen zwischen zusammengehörigen Paaren (KBG-VBG, KAA-VAA, KAV-VAV), was die Dialogstruktur widerspiegelt.
- Die Zentralitätsanalyse identifiziert KBBd und VBBd als wichtigste Knoten – dies entspricht der zentralen Rolle der Bedarfsermittlung in Verkaufsgesprächen.
- Die Attention-Analyse bestätigt die Markov-Eigenschaft: Der unmittelbare Vorgänger ist der wichtigste Prädiktor.

6.4 Grenzen

Die hybride Integration hat auch Grenzen:

- Die computerlinguistischen Verfahren wurden nicht auf den Originaldaten trainiert, sondern nutzen vortrainierte Modelle oder einfache Statistiken.
- Die Attention-Analyse ist vereinfacht und bildet nicht die komplexen Abhängigkeiten moderner Transformer ab.
- Die Ergebnisse sind deskriptiv und erlauben keine kausalen Schlüsse.

7 Fazit und Ausblick

Die hybride Integration computerlinguistischer Verfahren in die ARS 4.0 erweitert das Methodenspektrum um komplementäre Analyseperspektiven, ohne die methodologische Kontrolle zu gefährden. Die interpretativen Kategorien der ARS 3.0 bleiben die Grundlage – die neuen Verfahren dienen der Validierung, Visualisierung und vertieften Analyse.

Weiterführende Forschung könnte:

- **Erweiterte CRF-Modelle**: Integration von Embedding-Features
- **Dynamische Graphen**: Zeitliche Entwicklung der Grammatik-Struktur
- **Multilinguale Analyse**: Übertragung auf andere Sprachen
- **Interaktive Visualisierungen**: Webbasierte Exploration der Grammatik

Literatur

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT 2019*, 4171-4186.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. *Proceedings of ICML 2001*, 282-289.

Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *Proceedings of EMNLP-IJCNLP 2019*, 3982-3992.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1), 61-80.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. *Advances in Neural Information Processing Systems 30*, 5998-6008.

A Die acht Transkripte mit Terminalzeichen

A.1 Transkript 1 - Metzgerei

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

A.2 Transkript 2 - Marktplatz (Kirschen)

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

A.3 Transkript 3 - Fischstand

Terminalzeichenkette 3: KBBd, VBBd, VAA, KAA

A.4 Transkript 4 - Gemüsestand (ausführlich)

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

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

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

A.6 Transkript 6 - Käseverkaufsstand

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

A.7 Transkript 7 - Bonbonstand

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

A.8 Transkript 8 - Bäckerei

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