Between Interpretation and Computation

Didactic Exploration of Computational Linguistics Methods
with Augmented Transcripts of Sales Conversations

Paul Koop

Teaching and Learning Materials 2026

Abstract

This teaching and learning material serves the didactic exploration of computational linguistics methods based on eight transcripts of sales conversations. In contrast to the previous ARS versions 2.0 and 3.0, which were based on interpretively formed terminal symbols, this material takes the step toward automatic language processing. The methods are trained on augmented data for demonstration purposes to make their functioning transparent. The focus is on didactic knowledge acquisition, not on empirical validity. Scenarios C (Computational Linguistics Integration) and D (Hybrid Modeling) are developed step by step and compared with each other.

Contents

1 Introduction: Didactic Goals and Methodological Reflection 2

2 The Eight Transcripts: Raw Data and Terminal Symbols 3

2.1 The Raw Data . . . 3
2.1.1 Transcript 1 - Butcher Shop . . . 3
2.1.2 Transcript 2 - Market Square (Cherries) . . . 3
2.1.3 Transcript 3 - Fish Stall . . . 4
2.1.4 Transcript 4 - Vegetable Stall (Detailed) . . . 4
2.1.5 Transcript 5 - Vegetable Stall (with KAV at Beginning) . . . 4
2.1.6 Transcript 6 - Cheese Stand . . . 4
2.1.7 Transcript 7 - Candy Stall . . . 5
2.1.8 Transcript 8 - Bakery . . . 5
2.2 The Terminal Symbol Strings (ARS 3.0) . . . 6

3 Scenario C: Computational Linguistics Integration 7

3.1 Didactic Augmentation . . . 7
3.2 Speech Act Recognition with Transformer Models . . . 8
3.3 Word Embeddings and Semantic Similarity . . . 12
3.4 Topic Modeling with BERTopic . . . 15
3.5 Rhetorical Structure Theory (RST) . . . 18
3.6 Integration of Components in Scenario C . . . 23

4 Scenario D: Hybrid Modeling 28

4.1 CRF for Sequential Dependencies . . . 28
4.2 Transformer Embeddings as Supplement . . . 32
4.3 Graph Neural Networks for the Nonterminal Hierarchy . . . 35
4.4 Attention Mechanisms for Relevant Predecessors . . . 38
4.5 Integration of Components in Scenario D . . . 42

5 Comparison of Scenarios and Methodological Reflection 47

5.1 Comparison of Approaches . . . 47
5.2 Didactic Insights from Scenario C . . . 47
5.3 Didactic Insights from Scenario D . . . 47
5.4 Conclusion for Teaching Practice . . . 48

6 Outlook 48

1 Introduction: Didactic Goals and Methodological Reflection

The previous versions of Algorithmic Recursive Sequence Analysis (ARS 2.0 and 3.0) have shown how formal grammars can be induced from interpretively obtained terminal symbol strings. These methods remain methodologically controlled: the category formation occurs through qualitative interpretation, the formal models merely explicate the observable regularities.

The following scenarios C and D venture a step beyond this methodological boundary. They explore how computational linguistics methods – especially neural networks, word embeddings, and topic models – could be applied to the eight transcripts if they were augmented for demonstration purposes.

This document is conceived as teaching and learning material. It pursues the following didactic goals:

1. **Understanding neural architectures**: How do transformers, LSTM networks, and attention mechanisms work on sequence data?
2. **Data augmentation as a technique**: How can one handle small datasets to demonstrate the functioning of methods?
3. **Comparison of different modeling levels**: What differences exist between purely computational linguistics (C) and hybrid (D) approaches?
4. **Methodological reflection**: Where are the limits of automatic methods compared to interpretive category formation?

All implementations presented here work with augmented data – the eight original transcripts were artificially multiplied to enable the training of neural networks. The results are therefore not empirically valid but serve exclusively for didactic illustration.

2 The Eight Transcripts: Raw Data and Terminal Symbols

2.1 The Raw Data

The following eight transcripts document sales conversations at Aachen market square in June/July 1994. They form the empirical basis for all subsequent analyses.

2.1.1 Transcript 1 - Butcher Shop

Date: June 28, 1994, **Location:** Butcher Shop, Aachen, 11:00 AM

```
Customer: Good day!
Salesperson: Good day!
Customer: One of the coarse liver sausage, please.
Salesperson: How much would you like?
Customer: Two hundred grams.
Salesperson: Anything else?
Customer: Yes, then also a piece of the Black Forest ham.
Salesperson: How large should the piece be?
Customer: Around three hundred grams.
Salesperson: That will be eight marks twenty.
Customer: Here you go.
Salesperson: Thank you and have a nice day!
Customer: Thanks, you too!
```

Listing 1: Transcript 1 - Raw Data

2.1.2 Transcript 2 - Market Square (Cherries)

Date: June 28, 1994, **Location:** Market Square, Aachen

```
Seller: Everyone can try cherries here!
Customer 1: Half a kilo of cherries, please.
Seller: Half a kilo? Or one kilo?
Seller: Three marks, please.
Customer 1: Thank you very much!
Seller: Everyone can try cherries here!
Customer 2: Half a kilo, please.
Seller: Three marks, please.
Customer 2: Thank you very much!
```

Listing 2: Transcript 2 - Raw Data

2.1.3 Transcript 3 - Fish Stall

Date: June 28, 1994, **Location:** Fish Stall, Market Square, Aachen

```
Customer: One pound of saithe, please.
Seller: Saithe, all right.
Seller: Four marks nineteen, please.
Customer: Thank you very much!
```

Listing 3: Transcript 3 - Raw Data

2.1.4 Transcript 4 - Vegetable Stall (Detailed)

Date: June 28, 1994, **Location:** Vegetable Stall, Aachen, Market Square, 11:00 AM

```
Customer: Listen, I'll take some mushrooms with me.
Seller: Brown or white?
Customer: Let's take the white ones.
Seller: They're both fresh, don't worry.
Customer: What about chanterelles?
Seller: Ah, they're great!
Customer: Can I put them in rice salad?
Seller: Better to briefly saut them in a pan.
Customer: Okay, I'll do that.
Seller: Have a nice day!
Customer: Likewise!
```

Listing 4: Transcript 4 - Raw Data

2.1.5 Transcript 5 - Vegetable Stall (with KAV at Beginning)

Date: June 26, 1994, **Location:** Vegetable Stall, Aachen, Market Square, 11:00 AM

```
Customer 1: Goodbye!
Customer 2: I would like a kilo of the Granny Smith apples here.
Seller: Anything else?
Customer 2: Yes, another kilo of onions.
Seller: Six marks twenty-five, please.
Customer 2: Goodbye!
```

Listing 5: Transcript 5 - Raw Data

2.1.6 Transcript 6 - Cheese Stand

Date: June 28, 1994, **Location:** Cheese Stand, Aachen, Market Square

```
Customer 1: Good morning!
Seller: Good morning!
Customer 1: I would like five hundred grams of Dutch Gouda.
Seller: In one piece?
Customer 1: Yes, in one piece, please.
```

Listing 6: Transcript 6 - Raw Data

2.1.7 Transcript 7 - Candy Stall

Date: June 28, 1994, **Location:** Candy Stall, Aachen, Market Square, 11:30 AM

```
Customer: I would like one hundred grams of the assorted ones.
Seller: For home or to take away?
Customer: To take away, please.
Seller: Fifty pfennigs, please.
Customer: Thanks!
```

Listing 7: Transcript 7 - Raw Data

2.1.8 Transcript 8 - Bakery

Date: July 9, 1994, **Location:** Bakery, Aachen, 12:00 PM

```
(Footsteps audible, background noises, partially unintelligible)
Customer: Good day!
(Unintelligible greeting in the background)
Salesperson: One of our best coffee, freshly ground, please.
(Noises of coffee grinder, packaging sounds)
Salesperson: Anything else?
Customer: Yes, also two pieces of fruit salad and a small bowl of
   cream.
Salesperson: All right!
(Noises of coffee grinder, paper sounds)
Salesperson: A small bowl of cream, yes?
Customer: Yes, thanks.
(Door noise, laughter, paper sounds)
Salesperson: Nobody takes care of oiling the doors.
Customer: Yes, that's always the case.
(Laughter, sounds of coins and packaging)
Salesperson: That will be fourteen marks and nineteen pfennigs,
   please.
Customer: I'll pay in small change.
(Laughter and sounds of coins)
Salesperson: Thank you very much, have a nice Sunday!
```

```
Customer: Thanks, you too!
```

Listing 8: Transcript 8 - Raw Data

2.2 The Terminal Symbol Strings (ARS 3.0)

For ARS 3.0, these raw data were converted into terminal symbol strings, which served as the basis for hierarchical grammar induction:

Table 1: Terminal Symbol Strings of the Eight Transcripts

Transcript	**Terminal Symbol String**
1 (Butcher)	KBG, VBG, KBBd, VBBd, KBA, VBA, KBBd, VBBd, KBA, VAA, KAA, VA
2 (Cherries)	VBG, KBBd, VBBd, VAA, KAA, VBG, KBBd, VAA, KAA
3 (Fish)	KBBd, VBBd, VAA, KAA
4 (Vegetable)	KBBd, VBBd, KBA, VBA, KBBd, VBA, KAE, VAE, KAA, VAV, KAV
5 (Vegetable KAV)	KAV, KBBd, VBBd, KBBd, VAA, KAV
6 (Cheese)	KBG, VBG, KBBd, VBBd, KAA
7 (Candy)	KBBd, VBBd, KBA, VAA, KAA
8 (Bakery)	KBG, VBBd, KBBd, VBA, VAA, KAA, VAV, KAV

The meaning of the terminal symbols:

- **KBG**: Customer greeting
- **VBG**: Salesperson greeting
- **KBBd**: Customer need (concrete)
- **VBBd**: Salesperson inquiry
- **KBA**: Customer response
- **VBA**: Salesperson reaction
- **KAE**: Customer inquiry
- **VAE**: Salesperson information
- **KAA**: Customer completion
- **VAA**: Salesperson completion
- **KAV**: Customer farewell
- **VAV**: Salesperson farewell

3 Scenario C: Computational Linguistics Integration

Scenario C implements a fully computational linguistics modeling of the eight transcripts. It comprises four components:

1. **Speech Act Recognition**: Automatic recognition of speech acts from raw data
2. **Word Embeddings**: Vector representations of utterances
3. **Topic Modeling**: Identification of thematic shifts
4. **Rhetorical Structure Theory (RST)**: Analysis of argumentative structure

3.1 Didactic Augmentation

Since neural networks require large amounts of data for training, the eight transcripts are augmented for demonstration purposes:

```python
def augment_transcripts_for_teaching(transcripts, factor=20):
    """
    Augments the eight transcripts for didactic purposes.

    Didactic note: This augmentation serves exclusively for illustrating
    the methodology. The resulting data are not empirically valid but
    merely enable demonstration of how neural methods function.
    """
    augmented = []

    # 1. Basic augmentation: simple copying
    for _ in range(factor):
        augmented.extend(transcripts)

    # 2. Syntactic variations (didactically controlled)
    import copy
    import random

    for transcript in transcripts:
```

```
        for _ in range(factor // 4):
            var = copy.deepcopy(transcript)
            # Swap two adjacent utterances (rarely)
            if len(var) > 3 and random.random() < 0.1:
                idx = random.randint(0, len(var)-2)
                var[idx], var[idx+1] = var[idx+1], var[idx]
            augmented.append(var)

    # 3. Lexical variations (synonyms)
    synonyms = {
        'Good day': ['Good morning', 'Hello', 'Good evening'],
        'Thanks': ['Thank you', 'Thank you very much', 'Merci'],
        'Please': ['Please', 'You're welcome']
    }

    # Further variations could be implemented here

    return augmented
```

Listing 9: Data Augmentation for Teaching Purposes

3.2 Speech Act Recognition with Transformer Models

Automatic recognition of speech acts is performed with a fine-tuned BERT model:

```
"""
Speech Act Recognition with transformer-based models
Didactic implementation for teaching purposes
"""

import torch
import torch.nn as nn
from transformers import BertTokenizer, BertModel
import numpy as np
from sklearn.preprocessing import LabelEncoder
from torch.utils.data import Dataset, DataLoader

class SpeechActDataset(Dataset):
    """Dataset for Speech Act Recognition"""
```

```python
    def __init__(self, utterances, labels, tokenizer, max_length=128):
        self.utterances = utterances
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_length = max_length

    def __len__(self):
        return len(self.utterances)

    def __getitem__(self, idx):
        utterance = self.utterances[idx]
        label = self.labels[idx]

        encoding = self.tokenizer(
            utterance,
            truncation=True,
            padding='max_length',
            max_length=self.max_length,
            return_tensors='pt'
        )

        return {
            'input_ids': encoding['input_ids'].flatten(),
            'attention_mask': encoding['attention_mask'].flatten(),
            'label': torch.tensor(label, dtype=torch.long)
        }

class BertSpeechActClassifier(nn.Module):
    """
    BERT-based classifier for speech acts
    Didactically simplified architecture
    """
    def __init__(self, num_classes=12, dropout=0.3):
        super().__init__()
        self.bert = BertModel.from_pretrained('bert-base-german-cased')
        self.dropout = nn.Dropout(dropout)
        self.classifier = nn.Linear(768, num_classes)
```

```python

        # Freeze BERT layers for didactic purposes (faster
            training)
        for param in self.bert.parameters():
            param.requires_grad = False

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids,
            attention_mask=attention_mask)
        pooled_output = outputs.pooler_output
        dropped = self.dropout(pooled_output)
        logits = self.classifier(dropped)
        return logits

def prepare_speech_act_data(transcripts, terminal_chains):
    """
    Prepares data for speech act training
    """
    utterances = []
    labels = []

    # Extract all utterances from raw data
    # Simplified: use terminal symbols directly for didactic
        purposes
    for trans, chain in zip(transcripts, terminal_chains):
        for symbol in chain:
            utterances.append(f"Example utterance for {symbol
                }")
            labels.append(symbol)

    # Label encoding
    label_encoder = LabelEncoder()
    y_encoded = label_encoder.fit_transform(labels)

    return utterances, y_encoded, label_encoder

def train_speech_act_model(utterances, labels, epochs=10):
    """
    Trains the speech act recognition model
    """
```

```
    tokenizer = BertTokenizer.from_pretrained('bert-base-german-cased')
    dataset = SpeechActDataset(utterances, labels, tokenizer)
    dataloader = DataLoader(dataset, batch_size=8, shuffle=True)

    model = BertSpeechActClassifier(num_classes=len(set(labels)))
    optimizer = torch.optim.Adam(model.parameters(), lr=2e-5)
    criterion = nn.CrossEntropyLoss()

    print("\n=== Speech Act Recognition Training (Didactic) ===")
    for epoch in range(epochs):
        total_loss = 0
        for batch in dataloader:
            optimizer.zero_grad()
            outputs = model(batch['input_ids'], batch['attention_mask'])
            loss = criterion(outputs, batch['label'])
            loss.backward()
            optimizer.step()
            total_loss += loss.item()

        print(f"Epoch {epoch+1}: Loss = {total_loss/len(dataloader):.4f}")

    return model, tokenizer, label_encoder

# Didactic note
print("\n" + "="*70)
print("DIDACTIC NOTE ON SPEECH ACT RECOGNITION")
print("="*70)
print("The implementation shown here uses augmented")
print("data and serves exclusively teaching purposes.")
print("Automatic recognition of speech acts would in practice:")
print("      Require millions of annotated training data")
print("      Be fine-tuned to specific domains (sales conversations)")
```

```python
print("        Be subject to considerable uncertainties")
```

Listing 10: Speech Act Recognition with BERT

3.3 Word Embeddings and Semantic Similarity

For quantifying semantic similarity, pre-trained word embeddings are used:

```python
"""
Word Embeddings for Semantic Similarity Analysis
Didactic implementation with pre-trained models
"""

from sentence_transformers import SentenceTransformer
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt
import seaborn as sns

class SemanticAnalyzer:
    """
    Analyzes semantic similarities between utterances
    """
    def __init__(self, model_name='paraphrase-multilingual-
       MiniLM-L12-v2'):
        print(f"Loading pre-trained model: {model_name}")
        self.model = SentenceTransformer(model_name)
        self.embeddings = {}

    def encode_utterances(self, utterances):
        """
        Creates embeddings for a list of utterances
        """
        embeddings = self.model.encode(utterances)
        for utt, emb in zip(utterances, embeddings):
            self.embeddings[utt] = emb
        return embeddings

    def similarity_matrix(self, utterances):
        """
        Calculates similarity matrix for all utterances
```

```
        """
        embeddings = self.encode_utterances(utterances)
        sim_matrix = cosine_similarity(embeddings)
        return sim_matrix

    def find_similar(self, query, utterances, top_k=5):
        """
        Finds the most similar utterances to a query
        """
        query_emb = self.model.encode([query])[0]
        utt_embs = self.encode_utterances(utterances)

        similarities = cosine_similarity([query_emb], utt_embs)[0]
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        results = []
        for idx in top_indices:
            results.append({
                'utterance': utterances[idx],
                'similarity': similarities[idx]
            })

        return results

    def visualize_similarity(self, utterances, labels=None):
        """
        Visualizes similarity matrix as heatmap
        """
        sim_matrix = self.similarity_matrix(utterances)

        plt.figure(figsize=(12, 10))
        sns.heatmap(sim_matrix,
                    xticklabels=labels if labels else range(len(utterances)),
                    yticklabels=labels if labels else range(len(utterances)),
                    cmap='viridis', vmin=0, vmax=1)
        plt.title('Semantic Similarity Between Utterances')
        plt.tight_layout()
```

```
        plt.savefig('semantic_similarity.png', dpi=150)
        plt.show()

# Didactic example
def demonstrate_semantic_analysis():
    """
    Demonstrates semantic analysis with examples
    """
    analyzer = SemanticAnalyzer()

    # Example utterances from the transcripts
    utterances = [
        "Good day!",
        "Good morning!",
        "One liver sausage, please.",
        "I would like sausage.",
        "Thank you!",
        "Thanks very much!",
        "Goodbye!",
        "Bye!"
    ]

    print("\n=== Semantic Similarity Analysis ===")

    # Calculate similarity matrix
    sim_matrix = analyzer.similarity_matrix(utterances)

    # Most similar utterances to "Good day!"
    similar = analyzer.find_similar("Good day!", utterances,
       top_k=3)
    print("\nMost similar to 'Good day!':")
    for r in similar:
        print(f"  {r['utterance']}: {r['similarity']:.3f}")

    # Visualization
    analyzer.visualize_similarity(utterances, utterances)

    return analyzer

# Didactic note
```

```
print("\n" + "="*70)
print("DIDACTIC NOTE ON WORD EMBEDDINGS")
print("="*70)
print("The embeddings used were pre-trained on large corpora"
    )
print("(Wikipedia, news, web texts). They capture general")
print("linguistic similarities, not the specific categories")
print("of sales conversations.")
```

Listing 11: Semantic Similarity with Word Embeddings

3.4 Topic Modeling with BERTopic

For identifying thematic shifts, BERTopic is used:

```
"""
Topic Modeling for Identifying Thematic Shifts
Didactic implementation with BERTopic
"""

from bertopic import BERTopic
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
import matplotlib.pyplot as plt

class TranscriptTopicModeler:
    """
    Performs topic modeling on the transcripts
    """
    def __init__(self):
        self.model = None
        self.topics = None
        self.probs = None

    def prepare_documents(self, transcripts):
        """
        Prepares transcripts as documents for topic modeling
        """
        documents = []
        metadata = []

```

```
        for i, transcript in enumerate(transcripts, 1):
            # Each transcript as one document
            doc = ' '.join(transcript)
            documents.append(doc)
            metadata.append(f'Transcript {i}')

        return documents, metadata

    def fit_model(self, documents):
        """
        Trains the topic model
        """
        # Custom stop words
        stopwords = ['please', 'thanks', 'thank', 'yes', 'no']
        vectorizer = CountVectorizer(stop_words=stopwords)

        self.model = BERTopic(
            embedding_model="paraphrase-multilingual-MiniLM-L12-v2",
            vectorizer_model=vectorizer,
            verbose=True,
            nr_topics='auto'
        )

        self.topics, self.probs = self.model.fit_transform(documents)
        return self.topics, self.probs

    def visualize_topics(self):
        """
        Visualizes the found topics
        """
        if self.model is None:
            return

        fig = self.model.visualize_topics()
        fig.write_html("topic_visualization.html")

        # Statistics
```

```python
        topic_counts = pd.Series(self.topics).value_counts()
        print("\n=== Topic Distribution ===")
        for topic, count in topic_counts.items():
            if topic == -1:
                print(f"Outlier: {count} documents")
            else:
                words = self.model.get_topic(topic)[:5]
                words_str = ', '.join([w for w, _ in words])
                print(f"Topic {topic}: {count} documents - {words_str}")

def demonstrate_topic_modeling(transcripts):
    """
    Demonstrates topic modeling on the transcripts
    """
    modeler = TranscriptTopicModeler()
    documents, metadata = modeler.prepare_documents(transcripts)

    print("\n=== Topic Modeling of Eight Transcripts ===")
    topics, probs = modeler.fit_model(documents)

    for i, (doc, topic, prob, meta) in enumerate(zip(documents, topics, probs, metadata)):
        if topic != -1:
            words = modeler.model.get_topic(topic)[:3]
            words_str = ', '.join([w for w, _ in words])
            print(f"{meta}: Topic {topic} (Confidence: {prob:.2f}) - {words_str}")
        else:
            print(f"{meta}: No clear topic (Outlier)")

    modeler.visualize_topics()
    return modeler

# Didactic note
print("\n" + "="*70)
print("DIDACTIC NOTE ON TOPIC MODELING")
print("="*70)
print("Topic modeling identifies latent themes in text
```

```
    corpora.")
print("With only eight documents, topic finding is unstable."
    )
print("The results therefore serve only to illustrate the")
print("methodology, not for substantive analysis.")
```

Listing 12: Topic Modeling with BERTopic

3.5 Rhetorical Structure Theory (RST)

For analyzing argumentative structure, an RST parser is implemented:

```
"""
Rhetorical Structure Theory (RST) Analysis
Didactic implementation for sequence data
"""

import networkx as nx
import matplotlib.pyplot as plt
from collections import defaultdict

class RSTRelation:
    """Represents an RST relation between text segments"""
    def __init__(self, type_name, nucleus, satellite,
        direction='nucleus-satellite'):
        self.type = type_name  # e.g., 'Elaboration', '
            Contrast', 'Cause'
        self.nucleus = nucleus  # Central segment
        self.satellite = satellite  # Supporting segment
        self.direction = direction

class SimpleRSTParser:
    """
    Simple RST parser for didactic purposes
    Based on cue phrases and structural patterns
    """

    # Cue phrases for different relations
    cue_phrases = {
        'Elaboration': ['for example', 'in particular', '
            namely', 'specifically'],
```

```
        'Contrast': ['but', 'however', 'on the other hand', '
            conversely'],
        'Cause': ['because', 'since', 'therefore', 'thus', '
            hence'],
        'Condition': ['if', 'provided that', 'as long as'],
        'Purpose': ['in order to', 'so that'],
        'Sequence': ['then', 'after that', 'first', 'finally'
            ]
    }

    def __init__(self):
        self.relations = []
        self.graph = nx.DiGraph()

    def segment_transcript(self, transcript):
        """
        Segments a transcript into elementary discourse units
            (EDUs)
        Simplified: each utterance is an EDU
        """
        return transcript

    def identify_relations(self, segments):
        """
        Identifies RST relations between segments
        Didactically simplified implementation
        """
        relations = []

        for i in range(len(segments)-1):
            current = segments[i]
            next_seg = segments[i+1]

            # Check for cue phrases
            for rel_type, cues in self.cue_phrases.items():
                for cue in cues:
                    if cue in current.lower() or cue in
                        next_seg.lower():
                        relations.append(RSTRelation(
                            type_name=rel_type,
```

```
                            nucleus=i,
                            satellite=i+1
                        ))
                        break

            # Default: Sequence relation
            if i < len(segments)-1:
                relations.append(RSTRelation(
                    type_name='Sequence',
                    nucleus=i,
                    satellite=i+1
                ))

        return relations

    def build_tree(self, segments, relations):
        """
        Builds an RST tree from identified relations
        """
        self.graph.clear()

        # Add nodes
        for i, seg in enumerate(segments):
            self.graph.add_node(i, text=seg[:30] + '...' if
               len(seg) > 30 else seg)

        # Add edges
        for rel in relations:
            self.graph.add_edge(rel.nucleus, rel.satellite,
                                relation=rel.type)

        return self.graph

    def parse(self, transcript):
        """
        Complete RST analysis of a transcript
        """
        segments = self.segment_transcript(transcript)
        relations = self.identify_relations(segments)
        tree = self.build_tree(segments, relations)
```

```

        return {
            'segments': segments,
            'relations': relations,
            'tree': tree
        }

    def visualize(self, title="RST Structure"):
        """
        Visualizes the RST tree
        """
        pos = nx.spring_layout(self.graph)
        plt.figure(figsize=(12, 8))

        # Draw nodes
        nx.draw_networkx_nodes(self.graph, pos, node_color='lightblue',
                                 node_size=500)

        # Draw edges with relation type as label
        for edge in self.graph.edges(data=True):
            nx.draw_networkx_edges(self.graph, pos, [(edge[0], edge[1])])
            nx.draw_networkx_edge_labels(
                self.graph, pos,
                {(edge[0], edge[1]): edge[2]['relation']}
            )

        # Node labels
        labels = {node: f"{node}: {self.graph.nodes[node]['text']}"
                  for node in self.graph.nodes()}
        nx.draw_networkx_labels(self.graph, pos, labels, font_size=8)

        plt.title(title)
        plt.axis('off')
        plt.tight_layout()
        plt.savefig('rst_structure.png', dpi=150)
        plt.show()
```

```python

def demonstrate_rst_analysis(transcripts):
    """
    Demonstrates RST analysis on the transcripts
    """
    parser = SimpleRSTParser()

    print("\n=== RST Analysis of Transcripts ===")

    for i, transcript in enumerate(transcripts, 1):
        print(f"\nTranscript {i}:")
        result = parser.parse(transcript)

        # Show identified relations
        for rel in result['relations'][:5]:  # Only first 5
            seg1 = result['segments'][rel.nucleus][:20] + '...'
            seg2 = result['segments'][rel.satellite][:20] + '...'
            print(f"  {rel.type}: {seg1}     {seg2}")

        if i == 1:  # Visualize only first transcript
            parser.visualize(f"RST Structure Transcript {i}")

    return parser

# Didactic note
print("\n" + "="*70)
print("DIDACTIC NOTE ON RST ANALYSIS")
print("="*70)
print("The RST analysis implemented here is greatly simplified.")
print("A full RST parser would:")
print("      Require extensive manual annotation")
print("      Work with trained neural models")
print("      Consider multiple hierarchy levels of discourse relations")
```

Listing 13: Rhetorical Structure Theory Parser

3.6 Integration of Components in Scenario C

The complete integration of all components in Scenario C:

```python
"""
Scenario C: Complete Computational Linguistics Integration
Didactic implementation for teaching purposes
"""

import os
import json
from datetime import datetime

class ScenarioC:
    """
    Integrates all computational linguistics components:
    - Speech Act Recognition
    - Word Embeddings / Semantic Analysis
    - Topic Modeling
    - RST Analysis
    """

    def __init__(self, transcripts, terminal_chains):
        self.transcripts = transcripts
        self.terminal_chains = terminal_chains
        self.results = {}

        print("\n" + "="*70)
        print("SCENARIO C: COMPUTATIONAL LINGUISTICS INTEGRATION")
        print("="*70)
        print("\nThis scenario demonstrates the application of")
        print("computational linguistics methods to the eight ")
        print("transcripts. All results serve didactic purposes")
        print("and are not empirically valid.\n")

    def run_speech_act_recognition(self):
        """
```

```
        Runs speech act recognition
        """
        print("\n--- Speech Act Recognition ---")
        utterances, labels, encoder = prepare_speech_act_data
            (
            self.transcripts, self.terminal_chains
        )

        model, tokenizer, label_encoder =
            train_speech_act_model(
            utterances, labels, epochs=5
        )

        self.results['speech_act'] = {
            'model': model,
            'tokenizer': tokenizer,
            'label_encoder': label_encoder,
            'num_classes': len(label_encoder.classes_)
        }

        return self.results['speech_act']

    def run_semantic_analysis(self):
        """
        Runs semantic similarity analysis
        """
        print("\n--- Semantic Similarity Analysis ---")
        analyzer = SemanticAnalyzer()

        # Collect all utterances
        all_utterances = []
        for transcript in self.transcripts:
            all_utterances.extend(transcript)

        # Similarity matrix
        sim_matrix = analyzer.similarity_matrix(
            all_utterances[:20])  # Only first 20

        self.results['semantic'] = {
            'analyzer': analyzer,
```

```
            'utterances': all_utterances,
            'similarity_matrix': sim_matrix
        }

        return self.results['semantic']

    def run_topic_modeling(self):
        """
        Runs topic modeling
        """
        print("\n--- Topic Modeling ---")
        modeler = TranscriptTopicModeler()
        documents, metadata = modeler.prepare_documents(self.transcripts)
        topics, probs = modeler.fit_model(documents)
        modeler.visualize_topics()

        self.results['topic'] = {
            'modeler': modeler,
            'topics': topics,
            'probabilities': probs,
            'documents': documents,
            'metadata': metadata
        }

        return self.results['topic']

    def run_rst_analysis(self):
        """
        Runs RST analysis
        """
        print("\n--- RST Analysis ---")
        parser = SimpleRSTParser()

        rst_results = []
        for i, transcript in enumerate(self.transcripts, 1):
            result = parser.parse(transcript)
            rst_results.append({
                'transcript_id': i,
                'segments': result['segments'],
```

```
                'relations': [(r.type, r.nucleus, r.satellite) for r in result['relations']]
            })

            if i == 1:
                parser.visualize(f"RST Structure Transcript {i}")

        self.results['rst'] = rst_results
        return rst_results

    def run_all(self):
        """
        Runs all analyses
        """
        self.run_speech_act_recognition()
        self.run_semantic_analysis()
        self.run_topic_modeling()
        self.run_rst_analysis()

        # Summary
        print("\n" + "="*70)
        print("SCENARIO C SUMMARY")
        print("="*70)
        print(f"    Speech Act Recognition: {self.results['speech_act']['num_classes']} classes")
        print(f"    Semantic Analysis: {len(self.results['semantic']['utterances'])} utterances")
        print(f"    Topic Modeling: {len(set(self.results['topic']['topics']))} topics")
        print(f"    RST Analysis: {len(self.results['rst'])} transcripts analyzed")

        return self.results

# Didactic execution
def run_scenario_c_demonstration():
    """
    Runs the complete demonstration of Scenario C
    """
```

```
    # Load transcripts
    from ars_data import transcripts, terminal_chains

    # Augment data for didactic purposes
    augmented_transcripts = augment_transcripts_for_teaching(transcripts, factor=10)
    augmented_chains = augment_transcripts_for_teaching(terminal_chains, factor=10)

    print("\n" + "="*70)
    print("DIDACTIC AUGMENTATION")
    print("="*70)
    print(f"Original: {len(transcripts)} transcripts")
    print(f"Augmented: {len(augmented_transcripts)} transcripts")

    # Run Scenario C
    scenario = ScenarioC(augmented_transcripts, augmented_chains)
    results = scenario.run_all()

    # Save results
    with open('scenario_c_results.json', 'w') as f:
        # Convert non-serializable objects
        serializable = {
            'speech_act': {'num_classes': results['speech_act']['num_classes']},
            'semantic': {'num_utterances': len(results['semantic']['utterances'])},
            'topic': {'num_topics': len(set(results['topic']['topics']))},
            'rst': results['rst']
        }
        json.dump(serializable, f, indent=2)

    print("\nResults saved to 'scenario_c_results.json'")

    return results

if __name__ == "__main__":
```

```
    run_scenario_c_demonstration()
```

Listing 14: Scenario C - Complete Integration

4 Scenario D: Hybrid Modeling

Scenario D integrates computational linguistics methods with the interpretively formed categories of ARS 3.0. It skips the complete automation of category formation (Scenario C) and uses the new methods complementarily.

4.1 CRF for Sequential Dependencies

Conditional Random Fields model dependencies of speech acts on the wider context:

```
"""
Conditional Random Fields (CRF) for Sequential Dependencies
Didactic implementation with sklearn-crfsuite
"""

import sklearn_crfsuite
from sklearn_crfsuite import metrics
import numpy as np

class CRFSequenceModel:
    """
    CRF model for sequence modeling of terminal symbols
    """

    def __init__(self):
        self.crf = sklearn_crfsuite.CRF(
            algorithm='lbfgs',
            c1=0.1,  # L1 regularization
            c2=0.1,  # L2 regularization
            max_iterations=100,
            all_possible_transitions=True
        )
        self.label_encoder = None

    def word2features(self, tokens, i):
        """
```

```
        Creates features for position i in the sequence
        """
        word = tokens[i]

        features = {
            'bias': 1.0,
            'word': word,
            'word.is_first': i == 0,
            'word.is_last': i == len(tokens) - 1,
            'word.prefix_K': word.startswith('K'),
            'word.prefix_V': word.startswith('V'),
            'word.suffix_A': word.endswith('A'),
            'word.suffix_B': word.endswith('B'),
            'word.suffix_E': word.endswith('E'),
            'word.suffix_G': word.endswith('G'),
            'word.suffix_V': word.endswith('V'),
        }

        # Context features
        if i > 0:
            word_prev = tokens[i-1]
            features.update({
                '-1:word': word_prev,
                '-1:word.prefix_K': word_prev.startswith('K'),
                '-1:word.prefix_V': word_prev.startswith('V'),
                '-1:word.suffix_A': word_prev.endswith('A'),
            })
        else:
            features['BOS'] = True

        if i < len(tokens) - 1:
            word_next = tokens[i+1]
            features.update({
                '+1:word': word_next,
                '+1:word.prefix_K': word_next.startswith('K'),
                '+1:word.prefix_V': word_next.startswith('V'),
```

```
                '+1:word.suffix_A': word_next.endswith('A'),
            })
        else:
            features['EOS'] = True

        return features

    def extract_features(self, sequences):
        """
        Extracts features for all sequences
        """
        X = []
        for seq in sequences:
            X.append([self.word2features(seq, i) for i in range(len(seq))])
        return X

    def fit(self, sequences, labels):
        """
        Trains the CRF model
        """
        X = self.extract_features(sequences)
        self.crf.fit(X, labels)
        return self

    def predict(self, sequences):
        """
        Predicts labels for new sequences
        """
        X = self.extract_features(sequences)
        return self.crf.predict(X)

    def evaluate(self, test_sequences, test_labels):
        """
        Evaluates the model
        """
        pred = self.predict(test_sequences)

        # Flatten for metrics
        y_true = [label for seq in test_labels for label in
```

```
            seq]
        y_pred = [label for seq in pred for label in seq]

        return {
            'accuracy': np.mean(np.array(y_true) == np.array(
                y_pred)),
            'classification_report': metrics.
                flat_classification_report(
                test_labels, pred, labels=sorted(set(y_true))
            )
        }

def demonstrate_crf(terminal_chains):
    """
    Demonstrates CRF modeling on terminal symbols
    """
    print("\n=== CRF Modeling of Terminal Symbols ===")

    # Train-test split (didactic)
    train_size = int(len(terminal_chains) * 0.7)
    train_chains = terminal_chains[:train_size]
    test_chains = terminal_chains[train_size:]

    # Extract features
    model = CRFSequenceModel()
    X_train = model.extract_features(train_chains)

    # Training
    print(f"Training CRF with {len(train_chains)} sequences
        ...")
    model.fit(train_chains, train_chains)  # Labels are the
        sequences themselves

    # Evaluation
    results = model.evaluate(test_chains, test_chains)
    print(f"\nAccuracy: {results['accuracy']:.3f}")

    return model
```

Listing 15: CRF for Sequential Dependencies

4.2 Transformer Embeddings as Supplement

Transformer embeddings are used in addition to categorical terminal symbols:

```python
"""
Transformer Embeddings as Supplement to Categorical Terminal Symbols
"""

import torch
import numpy as np
from sentence_transformers import SentenceTransformer

class TerminalEmbeddingEnricher:
    """
    Enriches terminal symbols with semantic embeddings of underlying utterances
    """

    def __init__(self, model_name='paraphrase-multilingual-MiniLM-L12-v2'):
        self.model = SentenceTransformer(model_name)
        self.symbol_to_embedding = {}
        self.symbol_to_text = self._create_symbol_mapping()

    def _create_symbol_mapping(self):
        """
        Creates a mapping from terminal symbols to example texts
        """
        return {
            'KBG': ['Good day', 'Good morning', 'Hello'],
            'VBG': ['Good day', 'Good morning', 'Hello back'],
            'KBBd': ['One liver sausage', 'I would like cheese', 'One kilo of apples please'],
            'VBBd': ['How much would you like?', 'Which kind?', 'Anything else?'],
            'KBA': ['Two hundred grams', 'The white ones please', 'Yes, please'],
            'VBA': ['All right', 'Coming right up', 'Okay'],
```

```
            'KAE': ['Can I put that in salad?', 'Where are these from?', 'Is it fresh?'],
            'VAE': ['Better to s a u t ', 'From the region', 'Yes, very fresh'],
            'KAA': ['Here you go', 'Thanks', 'Yes, thanks'],
            'VAA': ['That will be 8 marks 20', '3 marks please', '14 marks 19'],
            'KAV': ['Goodbye', 'Bye', 'Have a nice day'],
            'VAV': ['Thank you very much', 'Have a nice day', 'Goodbye']
        }

    def get_embedding(self, symbol):
        """
        Returns the embedding for a terminal symbol
        """
        if symbol in self.symbol_to_embedding:
            return self.symbol_to_embedding[symbol]

        # Average of example text embeddings
        texts = self.symbol_to_text.get(symbol, [symbol])
        embeddings = self.model.encode(texts)
        avg_embedding = np.mean(embeddings, axis=0)

        self.symbol_to_embedding[symbol] = avg_embedding
        return avg_embedding

    def enrich_sequence(self, sequence):
        """
        Enriches a sequence of terminal symbols with embeddings
        """
        symbols = sequence
        embeddings = np.array([self.get_embedding(sym) for sym in symbols])

        return {
            'symbols': symbols,
            'embeddings': embeddings,
            'combined': np.column_stack([
```

```
                self._one_hot_encode(symbols),
                embeddings
            ]) if len(symbols) > 0 else np.array([])
        }

    def _one_hot_encode(self, symbols):
        """
        One-hot encoding of terminal symbols
        """
        unique_symbols = sorted(set(self.symbol_to_text.keys()))
        symbol_to_idx = {sym: i for i, sym in enumerate(unique_symbols)}

        one_hot = np.zeros((len(symbols), len(unique_symbols)))
        for i, sym in enumerate(symbols):
            if sym in symbol_to_idx:
                one_hot[i, symbol_to_idx[sym]] = 1

        return one_hot

def demonstrate_embedding_enrichment():
    """
    Demonstrates enrichment of terminal symbols with embeddings
    """
    enricher = TerminalEmbeddingEnricher()

    print("\n=== Enrichment of Terminal Symbols with Embeddings ===")

    # Example sequence
    sequence = ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA']

    enriched = enricher.enrich_sequence(sequence)

    print(f"\nSequence: {'     '.join(sequence)}")
    print(f"Embedding dimension: {enriched['embeddings'].shape[1]}")
```

```
    print(f"One-hot dimension: {enriched['combined'].shape[1]
        - enriched['embeddings'].shape[1]}")
    print(f"Combined dimension: {enriched['combined'].shape
       [1]}")

    return enricher
```

Listing 16: Transformer Embeddings for Terminal Symbols

4.3 Graph Neural Networks for the Nonterminal Hierarchy

The nonterminal hierarchy is modeled as a Graph Neural Network:

```
"""
Graph Neural Network for the Nonterminal Hierarchy
Didactic implementation with PyTorch Geometric
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, GATConv
from torch_geometric.data import Data
import networkx as nx

class GrammarGNN(nn.Module):
    """
    Graph Neural Network for the grammar hierarchy
    """

    def __init__(self, input_dim, hidden_dim=64, num_classes
       =12):
        super().__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.classifier = nn.Linear(hidden_dim, num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
```

```python
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = self.classifier(x)
        return F.log_softmax(x, dim=1)

class GrammarHierarchyGNN:
    """
    Manages the GNN for the nonterminal hierarchy
    """

    def __init__(self, grammar_rules):
        self.grammar = grammar_rules
        self.graph = self._build_graph()
        self.model = None

    def _build_graph(self):
        """
        Builds a graph from the grammar hierarchy
        """
        G = nx.DiGraph()

        # Nodes: terminals and nonterminals
        all_symbols = set()

        # Nonterminals as nodes
        for nt, productions in self.grammar.items():
            all_symbols.add(nt)
            for prod, _ in productions:
                for sym in prod:
                    all_symbols.add(sym)

        # Edges: derivation relations
        for nt, productions in self.grammar.items():
            for prod, prob in productions:
                for sym in prod:
                    G.add_edge(nt, sym, weight=prob)

        return G

    def prepare_data(self):
```

```
        """
        Prepares data for the GNN
        """
        # Node indices
        nodes = list(self.graph.nodes())
        node_to_idx = {node: i for i, node in enumerate(nodes
           )}

        # Feature matrix (simplified: one-hot)
        x = torch.eye(len(nodes))

        # Edge index
        edge_index = []
        for u, v, data in self.graph.edges(data=True):
            edge_index.append([node_to_idx[u], node_to_idx[v
               ]])

        edge_index = torch.tensor(edge_index, dtype=torch.
           long).t().contiguous()

        return Data(x=x, edge_index=edge_index)

    def train(self, epochs=100):
        """
        Trains the GNN
        """
        data = self.prepare_data()
        self.model = GrammarGNN(input_dim=data.x.shape[1])

        optimizer = torch.optim.Adam(self.model.parameters(),
            lr=0.01)

        print("\n=== Training Grammar GNN ===")
        for epoch in range(epochs):
            self.model.train()
            optimizer.zero_grad()
            out = self.model(data.x, data.edge_index)

            # Self-supervised learning: graph reconstruction
            # Simplified: predict neighbors
```

```
                loss = F.nll_loss(out[data.edge_index[0]], data.
                    edge_index[1])

                loss.backward()
                optimizer.step()

                if epoch % 20 == 0:
                    print(f"Epoch {epoch}: Loss = {loss.item():.4
                        f}")

        return self.model

def demonstrate_gnn(grammar_rules):
    """
    Demonstrates GNN for the grammar hierarchy
    """
    print("\n=== Graph Neural Network for Nonterminal
        Hierarchy ===")

    gnn = GrammarHierarchyGNN(grammar_rules)
    print(f"Graph: {gnn.graph.number_of_nodes()} nodes, "
          f"{gnn.graph.number_of_edges()} edges")

    model = gnn.train(epochs=100)

    return gnn, model
```

Listing 17: Graph Neural Network for Nonterminal Hierarchy

4.4 Attention Mechanisms for Relevant Predecessors

Attention mechanisms identify particularly relevant predecessors for current decisions:

```
"""
Attention Mechanisms for Identifying Relevant Predecessors
"""

import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
```

```python

class SequenceAttention(nn.Module):
    """
    Attention mechanism for sequence modeling
    """

    def __init__(self, embedding_dim, hidden_dim=64):
        super().__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim

        # Attention parameters
        self.W_q = nn.Linear(embedding_dim, hidden_dim, bias=False)
        self.W_k = nn.Linear(embedding_dim, hidden_dim, bias=False)
        self.W_v = nn.Linear(embedding_dim, hidden_dim, bias=False)
        self.scale = hidden_dim ** 0.5

    def forward(self, x, mask=None):
        """
        x: (seq_len, batch, embedding_dim)
        """
        # Compute Query, Key, Value
        Q = self.W_q(x)  # (seq_len, batch, hidden_dim)
        K = self.W_k(x)  # (seq_len, batch, hidden_dim)
        V = self.W_v(x)  # (seq_len, batch, hidden_dim)

        # Attention scores
        scores = torch.matmul(Q.transpose(0, 1), K.transpose(0, 1).transpose(1, 2))
        scores = scores / self.scale

        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        # Attention weights
        attention_weights = F.softmax(scores, dim=-1)

```

```
        # Weighted sum
        context = torch.matmul(attention_weights, V.transpose(0, 1))

        return context, attention_weights

class SymbolPredictorWithAttention(nn.Module):
    """
    Predicts the next symbol with attention on predecessors
    """

    def __init__(self, num_symbols, embedding_dim=50, hidden_dim=64):
        super().__init__()
        self.embedding = nn.Embedding(num_symbols, embedding_dim)
        self.attention = SequenceAttention(embedding_dim, hidden_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True)
        self.classifier = nn.Linear(hidden_dim + embedding_dim, num_symbols)

    def forward(self, x):
        """
        x: (batch, seq_len) with symbol indices
        """
        # Embeddings
        embedded = self.embedding(x)  # (batch, seq_len, embedding_dim)

        # LSTM for sequential dependencies
        lstm_out, (hidden, cell) = self.lstm(embedded)

        # Attention over the sequence
        # Transpose for attention (seq_len, batch, embedding_dim)
        context, attention_weights = self.attention(embedded.transpose(0, 1))

```

```
        # Combine last LSTM state with attention context
        last_hidden = hidden[-1]  # (batch, hidden_dim)
        last_context = context[-1]  # (batch, hidden_dim)

        # Prediction
        combined = torch.cat([last_hidden, last_context], dim=-1)
        logits = self.classifier(combined)

        return logits, attention_weights

def demonstrate_attention(terminal_chains, symbol_to_idx):
    """
    Demonstrates attention mechanisms on the sequences
    """
    print("\n=== Attention Mechanisms for Relevant Predecessors ===")

    # Prepare data
    sequences = []
    for chain in terminal_chains:
        seq = [symbol_to_idx[sym] for sym in chain]
        sequences.append(seq)

    # Padding for batch processing
    from torch.nn.utils.rnn import pad_sequence
    sequences_padded = pad_sequence([torch.tensor(seq) for seq in sequences],
                                    batch_first=True, padding_value=0)

    # Initialize model
    model = SymbolPredictorWithAttention(num_symbols=len(symbol_to_idx))

    # Forward pass
    logits, attention_weights = model(sequences_padded[:2])  # Only first 2 sequences

    print(f"\nInput shape: {sequences_padded[:2].shape}")
```

```
    print(f"Attention weights shape: {attention_weights.shape
        }")
    print(f"Logits shape: {logits.shape}")

    # Visualize attention weights
    plot_attention_weights(attention_weights[0].detach().
        numpy(),
                              sequences[0], sequences[0])

    return model

def plot_attention_weights(attention, source_tokens,
    target_tokens):
    """
    Visualizes attention weights as heatmap
    """
    import matplotlib.pyplot as plt
    import seaborn as sns

    plt.figure(figsize=(10, 8))
    sns.heatmap(attention[:len(target_tokens), :len(
        source_tokens)],
                xticklabels=source_tokens,
                yticklabels=target_tokens,
                cmap='viridis', annot=True, fmt='.2f')
    plt.title('Attention Weights Between Predecessors and
        Prediction')
    plt.xlabel('Predecessor Symbols')
    plt.ylabel('Prediction Position')
    plt.tight_layout()
    plt.savefig('attention_weights.png', dpi=150)
    plt.show()
```

Listing 18: Attention Mechanisms for Sequence Modeling

4.5 Integration of Components in Scenario D

The complete integration of all components in Scenario D:

```
"""
Scenario D: Hybrid Modeling
```

```python
Integration of computational linguistics methods with interpretive categories
"""

import json
import numpy as np

class ScenarioD:
    """
    Integrates computational linguistics methods complementarily to the
    interpretively formed categories of ARS 3.0
    """

    def __init__(self, terminal_chains, grammar_rules, reflection_log):
        self.terminal_chains = terminal_chains
        self.grammar_rules = grammar_rules
        self.reflection_log = reflection_log
        self.results = {}

        print("\n" + "="*70)
        print("SCENARIO D: HYBRID MODELING")
        print("="*70)
        print("\nThis scenario integrates computational linguistics")
        print("methods COMPLEMENTARILY to the interpretive")
        print("categories of ARS 3.0. The interpretive basis")
        print("is preserved but enriched by new methods.\n")

    def run_crf_modeling(self):
        """
        Runs CRF modeling on terminal symbols
        """
        print("\n--- CRF Modeling ---")
        crf_model = demonstrate_crf(self.terminal_chains)
        self.results['crf'] = {'model': crf_model}
        return crf_model

```

```python
    def run_embedding_enrichment(self):
        """
        Enriches terminal symbols with transformer embeddings
        """
        print("\n--- Embedding Enrichment ---")
        enricher = demonstrate_embedding_enrichment()

        # Example enriched sequence
        example_seq = self.terminal_chains[0][:5]
        enriched = enricher.enrich_sequence(example_seq)

        self.results['embeddings'] = {
            'enricher': enricher,
            'example': enriched
        }

        return enricher

    def run_gnn_hierarchy(self):
        """
        Models the nonterminal hierarchy as GNN
        """
        print("\n--- GNN for Nonterminal Hierarchy ---")
        gnn, model = demonstrate_gnn(self.grammar_rules)
        self.results['gnn'] = {'gnn': gnn, 'model': model}
        return gnn, model

    def run_attention_analysis(self):
        """
        Analyzes attention mechanisms on the sequences
        """
        print("\n--- Attention Analysis ---")

        # Symbol to index mapping
        all_symbols = set()
        for chain in self.terminal_chains:
            all_symbols.update(chain)
        symbol_to_idx = {sym: i for i, sym in enumerate(
            sorted(all_symbols))}

```

```
        model = demonstrate_attention(self.terminal_chains,
            symbol_to_idx)
        self.results['attention'] = {'model': model}

        return model

    def run_all(self):
        """
        Runs all analyses (complementary, not substitutive)
        """
        self.run_crf_modeling()
        self.run_embedding_enrichment()
        self.run_gnn_hierarchy()
        self.run_attention_analysis()

        # Summary
        print("\n" + "="*70)
        print("SCENARIO D SUMMARY")
        print("="*70)
        print("    CRF Modeling: Sequential dependencies
            modeled")
        print("    Embedding Enrichment: Terminal symbols
            semantically enriched")
        print("    GNN Hierarchy: Nonterminal structure as
            graph")
        print("    Attention Analysis: Relevant predecessors
            identified")
        print("\nThe interpretive categories of ARS 3.0
            remain")
        print("the foundation of all analyses. Computational"
            )
        print("linguistics methods serve complementary
            insight.")

        return self.results

def run_scenario_d_demonstration(terminal_chains,
    grammar_rules, reflection_log):
    """
    Runs the complete demonstration of Scenario D
```

```python
    """
    scenario = ScenarioD(terminal_chains, grammar_rules, reflection_log)
    results = scenario.run_all()

    # Save results
    with open('scenario_d_results.json', 'w') as f:
        # Simplified serializable version
        serializable = {
            'crf': {'status': 'completed'},
            'embeddings': {'status': 'completed'},
            'gnn': {'num_nodes': results['gnn'][0].graph.number_of_nodes()},
            'attention': {'status': 'completed'}
        }
        json.dump(serializable, f, indent=2)

    print("\nResults saved to 'scenario_d_results.json'")

    return results

# Didactic note
print("\n" + "="*70)
print("METHODOLOGICAL NOTE ON SCENARIO D")
print("="*70)
print("Scenario D preserves the interpretive basis of ARS 3.0.")
print("The computational linguistics methods are used COMPLEMENTARILY,")
print("not as a replacement for manual category formation.")
print("This corresponds to the methodological demand for")
print("control and transparency in the sense of XAI criteria.")
```

Listing 19: Scenario D - Complete Hybrid Integration

5 Comparison of Scenarios and Methodological Reflection

5.1 Comparison of Approaches

Table 2: Comparison of Scenarios C and D

Criterion	**Scenario C**	**Scenario D**
Category Formation	Automatic (Speech Act Recognition)	Interpretive (ARS 3.0)
Data Basis	Augmented raw data	Terminal symbol strings
Representation	Vector embeddings	Discrete categories + embeddings
Hierarchy	Automatically learned	Explicitly induced (ARS 3.0)
Transparency	Low (black box)	High (documented decisions)
Didactic Value	Functioning of neural methods	Integration of old and new methods
Empirical Validity	Not given	Limited (based on interpretation)
Methodological Control	Lost	Preserved

5.2 Didactic Insights from Scenario C

The implementation of Scenario C has shown:

1. **Need for large data volumes**: Neural methods require data volumes far exceeding the eight transcripts for valid results. Augmentation enables demonstration of functioning but does not replace real data.

2. **Opacity of decisions**: Automatically learned categories and attention weights are not easily comprehensible to third parties. The XAI criteria of meaningfulness and transparency are violated.

3. **Loss of interpretive basis**: Automatic speech act recognition does not capture the qualitatively meaningful distinctions of ARS (e.g., between KBA and KAA) but learns statistical correlations in vector space.

5.3 Didactic Insights from Scenario D

The implementation of Scenario D has shown:

1. **Complementarity instead of substitution**: Computational linguistics methods can provide valuable additional information (e.g., semantic similarities between different utterances) without replacing the interpretive basis.
2. **Validation possibilities**: Embedding similarities can be used to validate interpretive category formation: similar utterances should receive similar terminal symbols.
3. **Visualization of dependencies**: Attention mechanisms and CRF models visualize which predecessors are particularly relevant for current decisions – this can illustrate the sequential structure of conversations.
4. **Methodological control preserved**: Since interpretive categories form the foundation, all results remain tied back to qualitative decisions and are intersubjectively verifiable.

5.4 Conclusion for Teaching Practice

The didactic exploration of Scenarios C and D leads to the following conclusions:

1. **Scenario C is suitable for demonstrating the functioning** of neural methods but should be used with explicit note of lacking empirical validity and methodological problems.
2. **Scenario D is methodologically preferable** as it preserves the interpretive basis and uses computational linguistics methods complementarily. It conveys how old and new methods can be productively combined.
3. **Data augmentation is a valuable didactic tool** to demonstrate the functioning of methods with small datasets. The augmented nature of the data must always be made transparent.
4. **The XAI criteria** (meaningfulness, accuracy, knowledge limits) provide a suitable framework to evaluate different approaches and reflect on their strengths and weaknesses.

6 Outlook

The didactic implementations presented here can be further developed in several directions:

1. **Extension of augmentation strategies**: Beyond simple copying, more complex augmentations (paraphrasing, controlled variation) could be implemented.
2. **Integration of further methods**: e.g., Petri nets for concurrency, Bayesian networks for inference, or formal verification methods.
3. **Development of comparison metrics**: How can the results of different scenarios be compared quantitatively without losing the qualitative basis?
4. **Transfer to other datasets**: The methodology can be transferred to other interaction types (doctor-patient conversations, classroom interactions, etc.).

What remains crucial throughout is methodological control: the formal procedures must respect the interpretive character of the analysis and must not lead to its automation.

References

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.

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.

Flick, U. (2019). *Qualitative Social Research: An Introduction* (9th ed.). Rowohlt. [German original]

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

Mann, W. C., & Thompson, S. A. (1988). Rhetorical Structure Theory: Toward a functional theory of text organization. *Text*, 8(3), 243-281.

Przyborski, A., & Wohlrab-Sahr, M. (2021). *Qualitative Social Research: A Workbook* (5th ed.). De Gruyter Oldenbourg. [German original]

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

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.