Task: Conduct semantic drift verification, cultural-linguistic analysis and narrative remapping engine.

---

1. Objective

This Python script performs a comparative linguistic and sentiment analysis of national and propagandist narratives from different countries. It utilizes multilingual NLP tools to evaluate how narratives differ semantically and sentimentally across cultural contexts.

---

2. Tools and Libraries Used

- **sentence-transformers**: For generating multilingual sentence embeddings using the `paraphrase-multilingual-MiniLM-L12-v2` model.
- **transformers**: HuggingFace pipeline for sentiment analysis.
- **spaCy**: For natural language processing tasks (POS tagging and Named Entity Recognition) with English and Russian models.
- **scikit-learn**: Specifically `cosine_similarity` for semantic similarity comparison.
- **Google Colab Tools**: For optional file upload functionality.

---

3. How the Code Works

A. Model Setup and Data

1. **Model Initialization**:
   - Sentence embedding model: `paraphrase-multilingual-MiniLM-L12-v2`
   - Sentiment pipeline: HuggingFace's pre-trained sentiment classifier
   - spaCy NLP models for English (`en_core_web_sm`) and Russian (`ru_core_news_sm`) languages
2. **Narrative Definitions**:
   - A dictionary `narratives` holds sample statements from different countries including both neutral and propagandist variants.

B. Narrative Analysis Function

Function `analyze_narrative()` takes a text and language code:

- Generates **sentence embeddings**.
- Performs **sentiment analysis** (label and score).
- Extracts **POS tags** and **Named Entities** using spaCy.

C. Cross-Cultural Comparison

Function `compare_cross_culture("USA")` compares all other narratives to the USA's neutral narrative:

- Computes **cosine similarity** between vector embeddings.
- Displays **sentiment polarity**, **confidence score**, **POS tags**, and **entities** for each comparison.

---

4. Results and Insights

When comparing the U.S. narrative ("Freedom of speech is a fundamental right") to others:

- **Russia vs. USA**
  - **Propaganda_RU** shows low semantic similarity.
  - Sentiment may differ (likely negative or neutral depending on propagandist tone).
  - Lexical diversity through POS tags and fewer entities in propagandist content.
- **China vs. USA**
  - **Propaganda_CN** similarly deviates in meaning and sentiment.
  - POS tags and sentiment analysis likely reflect polarizing language in propagandist versions.
- **Sentiment Trend**:
  - Neutral narratives show **positive sentiment**.

    ○ Propaganda tends to lean towards **negative or fear-driven sentiment**, supporting the idea of emotional manipulation.

- **POS and NER**:
    ○ The propagandist texts tend to use fewer named entities and more emotional or abstract nouns/adjectives.

---

5. Conclusion

This code demonstrates a foundational framework for **semantic drift detection**, **cross-cultural discourse comparison**, and **narrative remapping**. It effectively combines machine learning and linguistic insights to:

- Identify how propaganda shifts narrative meaning.
- Analyze emotional tone across linguistic and cultural boundaries.
- Support multilingual information operations research.

```python
# semantic_drift_analyzer.py
"""
Semantic Drift, Cultural Linguistics, and Narrative Remapping Library

A comprehensive toolkit for analyzing semantic changes across time periods,
conducting cultural linguistic analysis, and implementing narrative remapping
techniques.
"""

import os
import re
import json
import numpy as np
import pandas as pd
from typing import List, Dict, Tuple, Optional, Union
from collections import Counter, defaultdict
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.decomposition import PCA, NMF
from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics.pairwise import cosine_similarity
import spacy
from gensim.models import Word2Vec, KeyedVectors
from gensim.models.phrases import Phrases, Phraser
```

```python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.sentiment import SentimentIntensityAnalyzer

# Download necessary NLTK resources
try:
   nltk.data.find('tokenizers/punkt')
   nltk.data.find('corpora/stopwords')
   nltk.data.find('sentiment/vader_lexicon.zip')
   nltk.data.find('corpora/wordnet')
except LookupError:
   nltk.download('punkt')
   nltk.download('stopwords')
   nltk.download('vader_lexicon')
   nltk.download('wordnet')

class TextProcessor:
   """Base class for text processing operations."""

   def __init__(self, language="english"):
       self.language = language
       self.nlp = spacy.load("en_core_web_md" if language == "english" else
language)
       self.stop_words = set(stopwords.words(language))
       self.lemmatizer = WordNetLemmatizer()

   def preprocess(self, text: str) -> str:
       """Preprocess text with basic cleaning operations."""
       # Convert to lowercase
       text = text.lower()
       # Remove special characters and digits
       text = re.sub(r'[^\w\s]', '', text)
       text = re.sub(r'\d+', '', text)
       # Remove extra whitespace
       text = re.sub(r'\s+', ' ', text).strip()
       return text

   def tokenize(self, text: str) -> List[str]:
       """Tokenize text into words."""
       return word_tokenize(text)

   def remove_stopwords(self, tokens: List[str]) -> List[str]:
       """Remove stopwords from a list of tokens."""
```

```
        return [token for token in tokens if token not in self.stop_words]

    def lemmatize(self, tokens: List[str]) -> List[str]:
        """Lemmatize tokens."""
        return [self.lemmatizer.lemmatize(token) for token in tokens]

    def get_named_entities(self, text: str) -> Dict[str, List[str]]:
        """Extract named entities from text."""
        doc = self.nlp(text)
        entities = defaultdict(list)
        for ent in doc.ents:
            entities[ent.label_].append(ent.text)
        return dict(entities)

    def full_process(self, text: str) -> List[str]:
        """Apply all preprocessing steps to text."""
        processed = self.preprocess(text)
        tokens = self.tokenize(processed)
        tokens = self.remove_stopwords(tokens)
        tokens = self.lemmatize(tokens)
        return tokens


class SemanticDriftAnalyzer(TextProcessor):
    """Analyze semantic drift across time periods in text corpora."""

    def __init__(self, language="english"):
        super().__init__(language)
        self.word_embeddings = {}
        self.time_periods = []

    def load_corpus(self,
                    corpus_data: Dict[str, List[str]],
                    time_periods: Optional[List[str]] = None) -> None:
        """
        Load corpus data by time period.

        Args:
            corpus_data: Dictionary with time periods as keys and lists of texts as
values
            time_periods: Optional list of time periods to process in order
        """
        if time_periods:
            self.time_periods = time_periods
        else:
```

```python
        self.time_periods = sorted(corpus_data.keys())

        self.corpus_data = {period: corpus_data[period] for period in
self.time_periods}

    def train_embeddings(self,
                        vector_size: int = 100,
                        window: int = 5,
                        min_count: int = 5,
                        workers: int = 4) -> None:
        """
        Train word embeddings for each time period.

        Args:
            vector_size: Dimensionality of word vectors
            window: Maximum distance between current and predicted word
            min_count: Ignore words with frequency below this
            workers: Number of threads to use
        """
        for period in self.time_periods:
            # Process each text in the time period
            processed_corpus = []
            for text in self.corpus_data[period]:
                processed_corpus.append(self.full_process(text))

            # Train phrase model to detect common phrases
            phrases = Phrases(processed_corpus, min_count=min_count)
            phraser = Phraser(phrases)

            # Apply phraser to the corpus
            phrased_corpus = [phraser[doc] for doc in processed_corpus]

            # Train Word2Vec model
            model = Word2Vec(sentences=phrased_corpus,
                           vector_size=vector_size,
                           window=window,
                           min_count=min_count,
                           workers=workers)

            self.word_embeddings[period] = model.wv

    def save_embeddings(self, directory: str) -> None:
        """Save trained embeddings to files."""
        os.makedirs(directory, exist_ok=True)
        for period, embeddings in self.word_embeddings.items():
```

```python
            filename = os.path.join(directory, f"{period}_embeddings.kv")
            embeddings.save(filename)

    def load_embeddings(self, directory: str) -> None:
        """Load embeddings from files."""
        for period in self.time_periods:
            filename = os.path.join(directory, f"{period}_embeddings.kv")
            self.word_embeddings[period] = KeyedVectors.load(filename)

    def get_semantic_neighbors(self,
                              word: str,
                              period: str,
                              n: int = 10) -> List[Tuple[str, float]]:
        """
        Get semantic neighbors of a word in a specific time period.

        Args:
            word: Target word
            period: Time period
            n: Number of neighbors to return

        Returns:
            List of (word, similarity) tuples
        """
        if period not in self.word_embeddings:
            raise ValueError(f"No embeddings for period: {period}")

        embeddings = self.word_embeddings[period]
        try:
            return embeddings.most_similar(word, topn=n)
        except KeyError:
            return []

    def track_word_trajectory(self,
                             word: str,
                             reference_words: List[str] = None,
                             n_neighbors: int = 5) -> Dict:
        """
        Track semantic trajectory of a word across time periods.

        Args:
            word: Target word to track
            reference_words: Optional list of words to compare against
            n_neighbors: Number of semantic neighbors to retrieve
```

```
        Returns:
            Dictionary with trajectory information
        """
        trajectory = {
            "word": word,
            "time_periods": self.time_periods,
            "neighbor_evolution": {},
            "similarity_to_reference": {},
            "present_in_periods": []
        }

        # Track neighbors evolution
        for period in self.time_periods:
            neighbors = self.get_semantic_neighbors(word, period, n_neighbors)
            if neighbors:
                trajectory["neighbor_evolution"][period] = neighbors
                trajectory["present_in_periods"].append(period)

        # Track similarity to reference words
        if reference_words:
            for ref_word in reference_words:
                trajectory["similarity_to_reference"][ref_word] = {}
                for period in trajectory["present_in_periods"]:
                    embeddings = self.word_embeddings[period]
                    try:
                        similarity = embeddings.similarity(word, ref_word)
                        trajectory["similarity_to_reference"][ref_word][period] =
similarity
                    except KeyError:
                        trajectory["similarity_to_reference"][ref_word][period] =
None

        return trajectory

    def visualize_semantic_drift(self,
                                 word: str,
                                 reference_words: List[str] = None,
                                 output_file: str = None) -> None:
        """
        Visualize semantic drift of a word across time periods.

        Args:
            word: Target word to track
            reference_words: Optional list of words to compare against
            output_file: Path to save visualization (if None, display interactively)
```

```
        """
        trajectory = self.track_word_trajectory(word, reference_words)

        plt.figure(figsize=(12, 8))

        # Plot similarity to reference words
        if reference_words:
            for ref_word in reference_words:
                if ref_word in trajectory["similarity_to_reference"]:
                    similarities = trajectory["similarity_to_reference"][ref_word]
                    periods = list(similarities.keys())
                    values = list(similarities.values())
                    valid_points = [(p, v) for p, v in zip(periods, values) if v is
not None]
                    if valid_points:
                        x_vals, y_vals = zip(*[(self.time_periods.index(p), v) for
p, v in valid_points])
                        plt.plot(x_vals, y_vals, 'o-', label=f"Similarity to
'{ref_word}'")

        plt.title(f"Semantic Drift of '{word}' Across Time Periods")
        plt.xlabel("Time Period")
        plt.xticks(range(len(self.time_periods)), self.time_periods, rotation=45)
        plt.ylabel("Semantic Similarity")
        plt.legend()
        plt.grid(True, linestyle='--', alpha=0.7)

        if output_file:
            plt.savefig(output_file, dpi=300, bbox_inches='tight')
        else:
            plt.tight_layout()
            plt.show()

    def detect_semantic_change_points(self,
                                      vocabulary: List[str],
                                      threshold: float = 0.2) -> Dict[str,
List[str]]:
        """
        Detect points of significant semantic change for a list of words.

        Args:
            vocabulary: List of words to analyze
            threshold: Similarity difference threshold to consider significant

        Returns:
```

```
            Dictionary of words and their change points
        """
        change_points = {}

        for word in vocabulary:
            word_changes = []
            for i in range(len(self.time_periods) - 1):
                p1 = self.time_periods[i]
                p2 = self.time_periods[i + 1]

                # Check if word exists in both periods
                if (p1 in self.word_embeddings and p2 in self.word_embeddings and
                    word in self.word_embeddings[p1] and word in
self.word_embeddings[p2]):

                    # Get neighbors in both time periods
                    neighbors_p1 = set(w for w, _ in
self.get_semantic_neighbors(word, p1, 20))
                    neighbors_p2 = set(w for w, _ in
self.get_semantic_neighbors(word, p2, 20))

                    # Calculate Jaccard distance between neighbor sets
                    intersection = len(neighbors_p1.intersection(neighbors_p2))
                    union = len(neighbors_p1.union(neighbors_p2))
                    jaccard_dist = 1 - (intersection / union if union > 0 else 0)

                    if jaccard_dist > threshold:
                        word_changes.append(f"{p1}->{p2}")

            if word_changes:
                change_points[word] = word_changes

        return change_points


class CulturalLinguisticAnalyzer(TextProcessor):
    """Analyze cultural linguistic patterns in text corpora."""

    def __init__(self, language="english"):
        super().__init__(language)
        self.sentiment_analyzer = SentimentIntensityAnalyzer()
        self.corpus = None
        self.corpus_metadata = None
        self.cultural_markers = {}
        self.cultural_clusters = None
```

```python
    def load_corpus_with_metadata(self,
                                 texts: List[str],
                                 metadata: List[Dict]) -> None:
        """
        Load corpus with associated metadata for cultural analysis.

        Args:
            texts: List of text documents
            metadata: List of metadata dictionaries for each text
        """
        assert len(texts) == len(metadata), "Texts and metadata must have same
length"
        self.corpus = texts
        self.corpus_metadata = metadata
        self.processed_corpus = [self.full_process(text) for text in texts]

    def define_cultural_markers(self, markers_dict: Dict[str, List[str]]) -> None:
        """
        Define cultural markers for analysis.

        Args:
            markers_dict: Dictionary with cultural categories as keys and lists of
                          terms/phrases as values
        """
        self.cultural_markers = markers_dict

    def extract_cultural_markers(self, text: str) -> Dict[str, int]:
        """
        Extract and count cultural markers in a text.

        Args:
            text: Input text

        Returns:
            Dictionary with marker categories and their frequencies
        """
        processed_text = " ".join(self.full_process(text))
        results = {}

        for category, terms in self.cultural_markers.items():
            count = 0
            for term in terms:
                # Count occurrences of each term
                pattern = r'\b' + re.escape(term.lower()) + r'\b'
```

```
                count += len(re.findall(pattern, processed_text))
            results[category] = count

        return results

    def analyze_cultural_distribution(self) -> pd.DataFrame:
        """
        Analyze cultural marker distribution across the corpus.

        Returns:
            DataFrame with cultural marker frequencies for each document
        """
        results = []

        for i, text in enumerate(self.corpus):
            # Extract cultural markers
            markers = self.extract_cultural_markers(text)

            # Combine with metadata
            document_data = {
                "doc_id": i,
                **self.corpus_metadata[i],
                **markers
            }
            results.append(document_data)

        return pd.DataFrame(results)

    def calculate_sentiment_by_culture(self,
                                       group_by: str = None) -> pd.DataFrame:
        """
        Calculate sentiment statistics grouped by cultural category.

        Args:
            group_by: Metadata field to group by (if None, analyze entire corpus)

        Returns:
            DataFrame with sentiment statistics by group
        """
        results = []

        for i, text in enumerate(self.corpus):
            # Calculate sentiment
            sentiment = self.sentiment_analyzer.polarity_scores(text)
```

```python
            # Extract cultural markers
            markers = self.extract_cultural_markers(text)

            # Create result entry
            entry = {
                "doc_id": i,
                "sentiment_pos": sentiment["pos"],
                "sentiment_neg": sentiment["neg"],
                "sentiment_neu": sentiment["neu"],
                "sentiment_compound": sentiment["compound"],
                **markers
            }

            # Add grouping field if specified
            if group_by and group_by in self.corpus_metadata[i]:
                entry["group"] = self.corpus_metadata[i][group_by]

            results.append(entry)

        df = pd.DataFrame(results)

        # Group by cultural category if specified
        if group_by:
            grouped = df.groupby("group").agg({
                "sentiment_pos": "mean",
                "sentiment_neg": "mean",
                "sentiment_neu": "mean",
                "sentiment_compound": "mean",
                **{cat: "mean" for cat in self.cultural_markers.keys()}
            }).reset_index()
            return grouped

        return df

    def cluster_cultural_patterns(self, n_clusters: int = 5) -> Dict:
        """
        Cluster documents based on cultural marker patterns.

        Args:
            n_clusters: Number of clusters to identify

        Returns:
            Dictionary with clustering results
        """
        # Extract cultural markers for all documents
```

```
        cultural_features = []
        for text in self.corpus:
            markers = self.extract_cultural_markers(text)
            cultural_features.append([markers[cat] for cat in
sorted(self.cultural_markers.keys())])

        # Normalize features
        cultural_features = np.array(cultural_features)
        if cultural_features.shape[0] > 0:  # Ensure we have data
            # Avoid division by zero by adding small constant
            feature_max = np.max(cultural_features, axis=0)
            feature_max = np.where(feature_max == 0, 1, feature_max)  # Replace
zeros with ones
            cultural_features = cultural_features / feature_max

        # Apply clustering
        kmeans = KMeans(n_clusters=min(n_clusters, len(self.corpus)),
random_state=42)
        cluster_labels = kmeans.fit_predict(cultural_features)

        # Prepare result
        cluster_centers = kmeans.cluster_centers_
        cluster_centers_dict = {}
        for i, center in enumerate(cluster_centers):
            cluster_centers_dict[f"cluster_{i}"] = {
                cat: center[j] for j, cat in
enumerate(sorted(self.cultural_markers.keys()))
            }

        self.cultural_clusters = {
            "labels": cluster_labels,
            "centers": cluster_centers_dict
        }

        return self.cultural_clusters

    def visualize_cultural_patterns(self,
                                  output_file: str = None,
                                  plot_type: str = "heatmap") -> None:
        """
        Visualize cultural patterns across the corpus.

        Args:
            output_file: Path to save visualization (if None, display interactively)
            plot_type: Type of plot ("heatmap", "cluster", "radar")
```

```
        """
        # Prepare data
        if not self.cultural_clusters:
            self.cluster_cultural_patterns()

        if plot_type == "heatmap":
            # Create cultural markers matrix
            cultural_data = []
            for text in self.corpus:
                markers = self.extract_cultural_markers(text)
                cultural_data.append([markers[cat] for cat in
sorted(self.cultural_markers.keys())])

            cultural_matrix = np.array(cultural_data)

            # Create heatmap
            plt.figure(figsize=(12, 10))
            plt.imshow(cultural_matrix, aspect='auto', cmap='viridis')
            plt.colorbar(label='Frequency')
            plt.xlabel('Cultural Categories')
            plt.ylabel('Documents')
            plt.title('Cultural Marker Heatmap')
            plt.xticks(range(len(self.cultural_markers)),
                      sorted(self.cultural_markers.keys()),
                      rotation=45, ha='right')

        elif plot_type == "cluster":
            # PCA for dimensionality reduction
            cultural_data = []
            for text in self.corpus:
                markers = self.extract_cultural_markers(text)
                cultural_data.append([markers[cat] for cat in
sorted(self.cultural_markers.keys())])

            pca = PCA(n_components=2)
            cultural_data_2d = pca.fit_transform(cultural_data)

            # Create scatter plot with cluster colors
            plt.figure(figsize=(10, 8))
            plt.scatter(cultural_data_2d[:, 0], cultural_data_2d[:, 1],
                       c=self.cultural_clusters["labels"], cmap='viridis',
                       alpha=0.7, s=50)
            plt.colorbar(label='Cluster')
            plt.xlabel('Principal Component 1')
            plt.ylabel('Principal Component 2')
```

```
        plt.title('Cultural Pattern Clusters')

    elif plot_type == "radar":
        # Create radar chart for cluster centers
        centers = self.cultural_clusters["centers"]
        categories = sorted(self.cultural_markers.keys())

        # Set up the radar chart
        angles = np.linspace(0, 2*np.pi, len(categories),
endpoint=False).tolist()
        angles += angles[:1]  # Close the circle

        fig, ax = plt.subplots(figsize=(10, 10), subplot_kw=dict(polar=True))

        for cluster_name, values in centers.items():
            values_list = [values[cat] for cat in categories]
            values_list += values_list[:1]  # Close the circle

            ax.plot(angles, values_list, linewidth=2, label=cluster_name)
            ax.fill(angles, values_list, alpha=0.1)

        ax.set_xticks(angles[:-1])
        ax.set_xticklabels(categories)
        ax.set_title('Cultural Cluster Patterns')
        plt.legend(loc='upper right')

    if output_file:
        plt.savefig(output_file, dpi=300, bbox_inches='tight')
    else:
        plt.tight_layout()
        plt.show()

def identify_cultural_crossovers(self, threshold: float = 0.5) -> List[Dict]:
    """
    Identify documents that exhibit significant cultural crossover.

    Args:
        threshold: Minimum normalized frequency to consider significant

    Returns:
        List of documents with significant cross-cultural patterns
    """
    crossovers = []

    for i, text in enumerate(self.corpus):
```

```
            # Extract cultural markers
            markers = self.extract_cultural_markers(text)

            # Normalize frequencies
            total = sum(markers.values())
            if total > 0:
                norm_markers = {k: v/total for k, v in markers.items()}

                # Find categories with significant presence
                significant_cats = [cat for cat, freq in norm_markers.items() if
freq >= threshold]

                if len(significant_cats) > 1:
                    crossovers.append({
                        "doc_id": i,
                        "metadata": self.corpus_metadata[i],
                        "significant_categories": significant_cats,
                        "normalized_frequencies": {cat: norm_markers[cat] for cat in
significant_cats}
                    })

        return crossovers


class NarrativeRemapper:
    """Analyze and transform narratives in text."""

    def __init__(self, language="english"):
        self.text_processor = TextProcessor(language)
        self.nlp = self.text_processor.nlp
        self.source_narrative = None
        self.target_narrative = None
        self.narrative_elements = ["entities", "themes", "sentiments", "structures"]
        self.transformation_rules = {}

    def analyze_narrative(self, text: str) -> Dict:
        """
        Analyze narrative elements in a text.

        Args:
            text: Input narrative text

        Returns:
            Dictionary with narrative analysis results
        """
```

```python
        doc = self.nlp(text)

        # Extract entities
        entities = self.text_processor.get_named_entities(text)

        # Extract themes (using key phrases and noun chunks)
        noun_chunks = [chunk.text for chunk in doc.noun_chunks]

        # Extract basic narratology elements
        sentences = [sent.text for sent in doc.sents]

        # Calculate sentiment by sentence
        sia = SentimentIntensityAnalyzer()
        sentiments = [sia.polarity_scores(sent) for sent in sentences]

        # Extract basic narrative structure
        narrative_units = []
        for i, sent in enumerate(sentences):
            narrative_units.append({
                "text": sent,
                "sentiment": sentiments[i],
                "entities": [ent.text for ent in doc.sents[i].ents],
                "position": i / len(sentences)  # Normalized position in narrative
            })

        # Identify potential turning points (significant sentiment shifts)
        turning_points = []
        for i in range(1, len(sentiments)):
            prev = sentiments[i-1]["compound"]
            curr = sentiments[i]["compound"]
            if abs(curr - prev) > 0.5:  # Significant sentiment shift
                turning_points.append(i)

        return {
            "entities": entities,
            "themes": noun_chunks,
            "sentiments": sentiments,
            "narrative_units": narrative_units,
            "turning_points": turning_points,
            "structure": {
                "sentence_count": len(sentences),
                "avg_sentence_length": sum(len(s.split()) for s in sentences) /
len(sentences),
                "overall_sentiment": np.mean([s["compound"] for s in sentiments])
            }
```

```python
    }

def set_source_narrative(self, text: str) -> None:
    """Set and analyze source narrative."""
    self.source_text = text
    self.source_narrative = self.analyze_narrative(text)

def set_target_narrative(self, text: str) -> None:
    """Set and analyze target narrative."""
    self.target_text = text
    self.target_narrative = self.analyze_narrative(text)

def define_transformation_rules(self, rules: Dict) -> None:
    """
    Define transformation rules for narrative remapping.

    Args:
        rules: Dictionary of transformation rules by category
    """
    self.transformation_rules = rules

def extract_transformation_rules(self) -> Dict:
    """
    Extract transformation rules by comparing source and target narratives.

    Returns:
        Dictionary of suggested transformation rules
    """
    if not self.source_narrative or not self.target_narrative:
        raise ValueError("Source and target narratives must be set")

    rules = {}

    # Entity transformations (map source entities to target entities)
    entity_map = {}
    source_entities = []
    for ent_type, ents in self.source_narrative["entities"].items():
        source_entities.extend(ents)

    target_entities = []
    for ent_type, ents in self.target_narrative["entities"].items():
        target_entities.extend(ents)

    # Simple heuristic: map entities by frequency rank
    source_entity_counts = Counter(source_entities)
```

```
        target_entity_counts = Counter(target_entities)

        source_top = [e for e, _ in source_entity_counts.most_common()]
        target_top = [e for e, _ in target_entity_counts.most_common()]

        for i in range(min(len(source_top), len(target_top))):
            entity_map[source_top[i]] = target_top[i]

        rules["entity_mappings"] = entity_map

        # Theme transformations
        source_themes = Counter(self.source_narrative["themes"]).most_common(10)
        target_themes = Counter(self.target_narrative["themes"]).most_common(10)

        theme_map = {}
        for i in range(min(len(source_themes), len(target_themes))):
            theme_map[source_themes[i][0]] = target_themes[i][0]

        rules["theme_mappings"] = theme_map

        # Sentiment transformations (overall trend)
        source_sentiment_trend = [s["compound"] for s in
self.source_narrative["sentiments"]]
        target_sentiment_trend = [s["compound"] for s in
self.target_narrative["sentiments"]]

        # Calculate sentiment transformation factor
        if len(source_sentiment_trend) > 0 and len(target_sentiment_trend) > 0:
            source_avg = np.mean(source_sentiment_trend)
            target_avg = np.mean(target_sentiment_trend)
            sentiment_shift = target_avg - source_avg
            rules["sentiment_transformation"] = {
                "shift": sentiment_shift,
                "amplify": 1.0 if abs(sentiment_shift) < 0.1 else abs(target_avg /
source_avg) if source_avg != 0 else 1.0
            }

        # Structure transformations
        rules["structure_transformation"] = {
            "pacing_factor": (self.target_narrative["structure"]["sentence_count"] /
                              self.source_narrative["structure"]["sentence_count"])
                              if self.source_narrative["structure"]["sentence_count"]
> 0 else 1.0,
            "complexity_factor":
(self.target_narrative["structure"]["avg_sentence_length"] /
```

```python
self.source_narrative["structure"]["avg_sentence_length"])
                                if
self.source_narrative["structure"]["avg_sentence_length"] > 0 else 1.0
        }

        return rules

    def remap_narrative(self,
                        input_text: str,
                        preserve_structure: bool = True) -> str:
        """
        Remap a narrative according to transformation rules.

        Args:
            input_text: Input narrative to transform
            preserve_structure: Whether to preserve original narrative structure

        Returns:
            Transformed narrative text
        """
        if not self.transformation_rules:
            self.transformation_rules = self.extract_transformation_rules()

        doc = self.nlp(input_text)
        sentences = [sent.text for sent in doc.sents]
        transformed_sentences = []

        # Apply entity replacements
        entity_map = self.transformation_rules.get("entity_mappings", {})

        for sentence in sentences:
            transformed = sentence

            # Replace entities
            for source_entity, target_entity in entity_map.items():
                pattern = re.compile(r'\b' + re.escape(source_entity) + r'\b',
re.IGNORECASE)
                transformed = pattern.sub(target_entity, transformed)

            # Apply theme transformations
            theme_map = self.transformation_rules.get("theme_mappings", {})
            for source_theme, target_theme in theme_map.items():
                pattern = re.compile(r'\b' + re.escape(source_theme) + r'\b',
re.IGNORECASE)
```

```
                transformed = pattern.sub(target_theme, transformed)

            transformed_sentences.append(transformed)

        # Apply structure transformation if needed
        if not preserve_structure:
            structure_rules =
self.transformation_rules.get("structure_transformation", {})
            pacing_factor = structure_rules.get("pacing_factor", 1.0)

            # Adjust number of sentences
            if pacing_factor < 1.0:  # Reduce number of sentences
                keep_indices = np.linspace(0, len(transformed_sentences)-1,
int(len(transformed_sentences) * pacing_factor))
                keep_indices = np.round(keep_indices).astype(int)
                transformed_sentences = [transformed_sentences[i] for i in
keep_indices]
            elif pacing_factor > 1.0:  # Increase number of sentences
                # Split some sentences to increase count
                new_sentences = []
                target_count = int(len(transformed_sentences) * pacing_factor)
                sentences_to_split = target_count - len(transformed_sentences)

                # Find sentences most suitable for splitting (longer ones)
                sentence_lengths = [len(s.split()) for s in transformed_sentences]
                split_candidates = np.argsort(sentence_lengths)[::-
1][:sentences_to_split]

                for i, sentence in enumerate(transformed_sentences):
                    if i in split_candidates:
                        # Simple split at conjunction or comma
                        conj_split = re.split(r'(, and |, but |, or |; )', sentence,
1)
                        if len(conj_split) > 1:
                            new_sentences.append(conj_split[0] +
conj_split[1].rstrip())
                            new_sentences.append(conj_split[2])
                        else:
                            comma_split = re.split(r', ', sentence, 1)
                            if len(comma_split) > 1:
                                new_sentences.append(comma_split[0] + '.')
                                new_sentences.append(comma_split[1])
                            else:
                                # Can't split nicely, just add as is
                                new_sentences.append(sentence)
```

```
                else:
                    new_sentences.append(sentence)

            transformed_sentences = new_sentences

        # Apply sentiment transformation
        sentiment_rules = self.transformation_rules.get("sentiment_transformation",
{})
        sentiment_shift = sentiment_rules.get("shift", 0)
        sentiment_amplify = sentiment_rules.get("amplify", 1.0)

        if abs(sentiment_shift) > 0.1 or abs(sentiment_amplify - 1.0) > 0.1:
            sia = SentimentIntensityAnalyzer()
            adjective_intensifiers = {
                "positive": {
                    "good": ["great", "excellent", "wonderful", "fantastic"],
                    "nice": ["delightful", "splendid", "marvelous"],
                    "happy": ["thrilled", "ecstatic", "overjoyed"],
                    "interesting": ["fascinating", "captivating", "enthralling"]
                },
                "negative": {
                    "bad": ["terrible", "awful", "dreadful", "horrendous"],
                    "sad": ["devastated", "heartbroken", "despondent"],
                    "angry": ["furious", "outraged", "enraged"],
                    "scary": ["terrifying", "horrifying", "nightmarish"]
                }
            }

            sentiment_adjusted = []
            for sentence in transformed_sentences:
                sent_score = sia.polarity_scores(sentence)["compound"]

                # Decide if we need to adjust this sentence
                target_score = sent_score * sentiment_amplify + sentiment_shift

                if abs(target_score - sent_score) > 0.3:  # Only adjust if
significant difference
                    # Find adjectives to replace
                    doc = self.nlp(sentence)
                    adj_replacements = []

                    for token in doc:
                        if token.pos_ == "ADJ":
                            adj_text = token.text.lower()
```

```
                            # Determine direction of adjustment
                            if target_score > sent_score:  # Make more positive
                                for adj, replacements in
adjective_intensifiers["positive"].items():
                                    if adj == adj_text or
token.similarity(self.nlp(adj)) > 0.6:
                                        replacement = np.random.choice(replacements)
                                        adj_replacements.append((token.text,
replacement))
                                        break

                                # Also weaken negative adjectives
                                for adj, replacements in
adjective_intensifiers["negative"].items():
                                    if adj == adj_text or
token.similarity(self.nlp(adj)) > 0.6:
                                        # Replace with milder form
                                        adj_replacements.append((token.text,
"somewhat " + token.text))
                                        break
                            else:  # Make more negative
                                for adj, replacements in
adjective_intensifiers["negative"].items():
                                    if adj == adj_text or
token.similarity(self.nlp(adj)) > 0.6:
                                        replacement = np.random.choice(replacements)
                                        adj_replacements.append((token.text,
replacement))
                                        break

                                # Also weaken positive adjectives
                                for adj, replacements in
adjective_intensifiers["positive"].items():
                                    if adj == adj_text or
token.similarity(self.nlp(adj)) > 0.6:
                                        # Replace with milder form
                                        adj_replacements.append((token.text,
"somewhat " + token.text))
                                        break

                    # Apply replacements
                    adjusted = sentence
                    for orig, repl in adj_replacements:
                        pattern = re.compile(r'\b' + re.escape(orig) + r'\b',
re.IGNORECASE)
```

```
                        adjusted = pattern.sub(repl, adjusted)

                    sentiment_adjusted.append(adjusted)
                else:
                    sentiment_adjusted.append(sentence)

            transformed_sentences = sentiment_adjusted

        # Recombine sentences
        return " ".join(transformed_sentences)

    def visualize_narrative_comparison(self, output_file: str = None) -> None:
        """
        Visualize comparison between source and target narratives.

        Args:
            output_file: Path to save visualization (if None, display interactively)
        """
        if not self.source_narrative or not self.target_narrative:
            raise ValueError("Source and target narratives must be set")

        # Prepare sentiment trajectories
        source_sentiments = [s["compound"] for s in
self.source_narrative["sentiments"]]
        source_x = np.linspace(0, 1, len(source_sentiments))

        target_sentiments = [s["compound"] for s in
self.target_narrative["sentiments"]]
        target_x = np.linspace(0, 1, len(target_sentiments))

        # Create plot
        plt.figure(figsize=(12, 8))

        # Plot sentiment trajectories
        plt.subplot(2, 1, 1)
        plt.plot(source_x, source_sentiments, 'b-', label='Source Narrative')
        plt.plot(target_x, target_sentiments, 'r-', label='Target Narrative')
        plt.axhline(y=0, color='k', linestyle='-', alpha=0.3)
        plt.ylim(-1.1, 1.1)
        plt.xlabel('Narrative Progress')
        plt.ylabel('Sentiment')
        plt.title('Narrative Sentiment Comparison')
        plt.legend()
        plt.grid(True, linestyle='--', alpha=0.7)
```

```
    # Plot entity comparison
    plt.subplot(2, 1, 2)

    # Combine entity types for comparison
    source_entity_counts = {}
    for ent_type, entities in self.source_narrative["entities"].items():
        for entity in entities:
            if entity in source_entity_counts:
                source_entity_counts[entity] += 1
            else:
                source_entity_counts[entity] = 1

    target_entity_counts = {}
    for ent_type, entities in self.target_narrative["entities"].items():
        for entity in entities:
            if entity in target_entity_counts:
                target_entity_counts[entity] += 1
            else:
                target_entity_counts[entity] = 1

    # Get top entities
    source_top = dict(Counter(source_entity_counts).most_common(5))
    target_top = dict(Counter(target_entity_counts).most_common(5))

    # Combine entities for chart
    all_entities = list(set(list(source_top.keys()) + list(target_top.keys())))

    # Create bar chart
    x = np.arange(len(all_entities))
    width = 0.35

    source_values = [source_top.get(entity, 0) for entity in all_entities]
    target_values = [target_top.get(entity, 0) for entity in all_entities]

    plt.bar(x - width/2, source_values, width, label='Source')
    plt.bar(x + width/2, target_values, width, label='Target')

    plt.xlabel('Entities')
    plt.ylabel('Frequency')
    plt.title('Top Entities Comparison')
    plt.xticks(x, all_entities, rotation=45, ha='right')
    plt.legend()

    plt.tight_layout()
```

```
        if output_file:
            plt.savefig(output_file, dpi=300, bbox_inches='tight')
        else:
            plt.show()


class SemDriftApp:
    """Application for analyzing semantic drift and cultural narrative mapping."""

    def __init__(self):
        """Initialize application components."""
        self.semantic_analyzer = SemanticDriftAnalyzer()
        self.cultural_analyzer = CulturalLinguisticAnalyzer()
        self.narrative_remapper = NarrativeRemapper()

    def load_data_from_directory(self,
                                directory: str,
                                time_period_regex: str = r'(\d{4})') -> Dict[str,
List[str]]:
        """
        Load corpus data from text files in directory, organizing by time periods.

        Args:
            directory: Path to directory containing text files
            time_period_regex: Regular expression to extract time period from
filename

        Returns:
            Dictionary with time periods as keys and lists of texts as values
        """
        corpus_data = defaultdict(list)

        for filename in os.listdir(directory):
            if filename.endswith('.txt'):
                # Extract time period from filename
                match = re.search(time_period_regex, filename)
                if match:
                    time_period = match.group(1)
                    filepath = os.path.join(directory, filename)

                    with open(filepath, 'r', encoding='utf-8') as f:
                        text = f.read()
                        corpus_data[time_period].append(text)

        return dict(corpus_data)
```

```
  def extract_corpus_metadata(self,
                              directory: str,
                              metadata_pattern: Dict[str, str] = None) ->
List[Dict]:
        """
        Extract metadata from text files for cultural analysis.

        Args:
            directory: Path to directory containing text files
            metadata_pattern: Dictionary mapping metadata fields to regex patterns

        Returns:
            List of metadata dictionaries for each text
        """
        metadata_list = []
        texts = []

        if metadata_pattern is None:
            # Default patterns
            metadata_pattern = {
                "year": r'Year:\s*(\d{4})',
                "author": r'Author:\s*([^\n]+)',
                "genre": r'Genre:\s*([^\n]+)',
                "region": r'Region:\s*([^\n]+)'
            }

        for filename in os.listdir(directory):
            if filename.endswith('.txt'):
                filepath = os.path.join(directory, filename)

                with open(filepath, 'r', encoding='utf-8') as f:
                    content = f.read()

                metadata = {"filename": filename}
                text_content = content

                # Extract metadata from content
                for field, pattern in metadata_pattern.items():
                    match = re.search(pattern, content)
                    if match:
                        metadata[field] = match.group(1)
                        # Remove metadata line from content
                        text_content = re.sub(pattern, '', text_content)
```

```
            metadata_list.append(metadata)
            texts.append(text_content)

        return texts, metadata_list

    def analyze_semantic_drift(self,
                              corpus_directory: str,
                              output_directory: str,
                              word_list: List[str] = None,
                              time_period_regex: str = r'(\d{4})') -> None:
        """
        Analyze semantic drift in corpus and generate visualizations.

        Args:
            corpus_directory: Directory containing corpus text files
            output_directory: Directory to save results
            word_list: List of words to analyze for semantic drift
            time_period_regex: Regular expression to extract time period from
filename
        """
        # Load corpus data
        corpus_data = self.load_data_from_directory(corpus_directory,
time_period_regex)
        self.semantic_analyzer.load_corpus(corpus_data)

        # Create output directory if it doesn't exist
        os.makedirs(output_directory, exist_ok=True)

        # Train word embeddings
        print("Training word embeddings...")
        self.semantic_analyzer.train_embeddings()

        # Save embeddings
        embeddings_dir = os.path.join(output_directory, "embeddings")
        self.semantic_analyzer.save_embeddings(embeddings_dir)

        # If no word list provided, extract common words
        if not word_list:
            print("Extracting common words...")
            all_texts = []
            for texts in corpus_data.values():
                all_texts.extend(texts)

            # Process texts to extract common words
            processor = TextProcessor()
```

```
            all_tokens = []
            for text in all_texts:
                tokens = processor.full_process(text)
                all_tokens.extend(tokens)

            # Get most common words
            word_counts = Counter(all_tokens)
            word_list = [word for word, count in word_counts.most_common(100)
                         if len(word) > 3]  # Filter out short words

        # Analyze semantic change points
        print("Analyzing semantic change points...")
        change_points =
self.semantic_analyzer.detect_semantic_change_points(word_list)

        # Save change points
        change_points_file = os.path.join(output_directory,
"semantic_change_points.json")
        with open(change_points_file, 'w', encoding='utf-8') as f:
            json.dump(change_points, f, indent=2)

        # Generate visualizations for words with detected changes
        print("Generating visualizations...")
        viz_dir = os.path.join(output_directory, "visualizations")
        os.makedirs(viz_dir, exist_ok=True)

        for word, periods in change_points.items():
            if periods:  # Only visualize words with detected changes
                output_file = os.path.join(viz_dir, f"{word}_drift.png")
                try:
                    self.semantic_analyzer.visualize_semantic_drift(word,
output_file=output_file)
                except Exception as e:
                    print(f"Error visualizing {word}: {e}")

        print(f"Analysis complete. Results saved to {output_directory}")

    def analyze_cultural_patterns(self,
                                 corpus_directory: str,
                                 output_directory: str,
                                 cultural_markers: Dict[str, List[str]] = None) ->
None:
        """
        Analyze cultural linguistic patterns in corpus.
```

```
        Args:
            corpus_directory: Directory containing corpus text files
            output_directory: Directory to save results
            cultural_markers: Dictionary of cultural markers to analyze
        """
        # Create output directory if it doesn't exist
        os.makedirs(output_directory, exist_ok=True)

        # Extract corpus texts with metadata
        texts, metadata = self.extract_corpus_metadata(corpus_directory)
        self.cultural_analyzer.load_corpus_with_metadata(texts, metadata)

        # Define default cultural markers if none provided
        if not cultural_markers:
            cultural_markers = {
                "individualism": ["I", "me", "my", "mine", "self", "personal",
"individual", "unique", "independent"],
                "collectivism": ["we", "us", "our", "ours", "community", "together",
"collective", "shared", "mutual"],
                "authority": ["authority", "obedience", "respect", "command",
"order", "tradition", "rule", "leader"],
                "equality": ["equal", "fair", "justice", "rights", "democracy",
"liberty", "freedom", "balance"],
                "progress": ["progress", "change", "innovation", "growth", "future",
"improve", "advance", "modern"],
                "tradition": ["tradition", "heritage", "custom", "ancestor", "old",
"preserve", "maintain", "history"]
            }

        self.cultural_analyzer.define_cultural_markers(cultural_markers)

        # Analyze cultural distribution
        print("Analyzing cultural distribution...")
        distribution = self.cultural_analyzer.analyze_cultural_distribution()

        # Save distribution data
        distribution_file = os.path.join(output_directory,
"cultural_distribution.csv")
        distribution.to_csv(distribution_file, index=False)

        # Calculate sentiment by cultural group
        print("Analyzing sentiment by cultural group...")
        if "genre" in distribution.columns:
            sentiment_by_genre =
self.cultural_analyzer.calculate_sentiment_by_culture(group_by="genre")
```

```
            sentiment_file = os.path.join(output_directory,
"sentiment_by_genre.csv")
            sentiment_by_genre.to_csv(sentiment_file, index=False)

        if "region" in distribution.columns:
            sentiment_by_region =
self.cultural_analyzer.calculate_sentiment_by_culture(group_by="region")
            sentiment_file = os.path.join(output_directory,
"sentiment_by_region.csv")
            sentiment_by_region.to_csv(sentiment_file, index=False)

        # Cluster cultural patterns
        print("Clustering cultural patterns...")
        clusters = self.cultural_analyzer.cluster_cultural_patterns()

        # Save cluster data
        clusters_file = os.path.join(output_directory, "cultural_clusters.json")
        with open(clusters_file, 'w', encoding='utf-8') as f:
            # Convert numpy arrays to lists for JSON serialization
            serializable_clusters = {
                "labels": clusters["labels"].tolist(),
                "centers": {k: {k2: float(v2) for k2, v2 in v.items()}
                           for k, v in clusters["centers"].items()}
            }
            json.dump(serializable_clusters, f, indent=2)

        # Generate visualizations
        print("Generating visualizations...")
        viz_dir = os.path.join(output_directory, "visualizations")
        os.makedirs(viz_dir, exist_ok=True)

        # Create different types of visualizations
        for plot_type in ["heatmap", "cluster", "radar"]:
            output_file = os.path.join(viz_dir, f"cultural_{plot_type}.png")
            try:

self.cultural_analyzer.visualize_cultural_patterns(output_file=output_file,

plot_type=plot_type)
            except Exception as e:
                print(f"Error generating {plot_type} visualization: {e}")

        # Identify cultural crossovers
        print("Identifying cultural crossovers...")
        crossovers = self.cultural_analyzer.identify_cultural_crossovers()
```

```python
        # Save crossover data
        crossovers_file = os.path.join(output_directory, "cultural_crossovers.json")
        with open(crossovers_file, 'w', encoding='utf-8') as f:
            # Convert metadata to serializable format
            serializable_crossovers = []
            for item in crossovers:
                item_copy = item.copy()
                item_copy["metadata"] = {k: str(v) for k, v in
item["metadata"].items()}
                serializable_crossovers.append(item_copy)
            json.dump(serializable_crossovers, f, indent=2)

        print(f"Cultural analysis complete. Results saved to {output_directory}")

    def remap_narrative(self,
                        source_file: str,
                        target_file: str,
                        input_file: str,
                        output_file: str,
                        visualization_file: str = None) -> None:
        """
        Remap a narrative based on source and target narratives.

        Args:
            source_file: Path to source narrative file
            target_file: Path to target narrative file
            input_file: Path to input narrative file to transform
            output_file: Path to save transformed narrative
            visualization_file: Optional path to save visualization
        """
        # Load source narrative
        with open(source_file, 'r', encoding='utf-8') as f:
            source_text = f.read()
        self.narrative_remapper.set_source_narrative(source_text)

        # Load target narrative
        with open(target_file, 'r', encoding='utf-8') as f:
            target_text = f.read()
        self.narrative_remapper.set_target_narrative(target_text)

        # Extract transformation rules
        print("Extracting transformation rules...")
        rules = self.narrative_remapper.extract_transformation_rules()
```

```
        # Save transformation rules
        rules_file = os.path.splitext(output_file)[0] + "_rules.json"
        with open(rules_file, 'w', encoding='utf-8') as f:
            # Convert any non-serializable values to strings
            serializable_rules = {}
            for category, rule_dict in rules.items():
                if isinstance(rule_dict, dict):
                    serializable_rules[category] = {k: str(v) if not isinstance(v,
(int, float, str, bool)) else v
                                                    for k, v in rule_dict.items()}
                else:
                    serializable_rules[category] = str(rule_dict)
            json.dump(serializable_rules, f, indent=2)

        # Load input narrative to transform
        with open(input_file, 'r', encoding='utf-8') as f:
            input_text = f.read()

        # Remap the narrative
        print("Remapping narrative...")
        transformed_text = self.narrative_remapper.remap_narrative(input_text)

        # Save the transformed narrative
        with open(output_file, 'w', encoding='utf-8') as f:
            f.write(transformed_text)

        # Generate visualization if requested
        if visualization_file:
            print("Generating visualization...")

self.narrative_remapper.visualize_narrative_comparison(visualization_file)

        print(f"Narrative remapping complete. Transformed narrative saved to
{output_file}")


def main():
    """Example usage of the semantic drift, cultural linguistics, and narrative
remapping toolkit."""
    # Create application
    app = SemDriftApp()

    # Example semantic drift analysis
    # app.analyze_semantic_drift("corpora/historical_texts",
"results/semantic_drift")
```

```
    # Example cultural pattern analysis
    # app.analyze_cultural_patterns("corpora/fiction_corpus",
"results/cultural_patterns")

    # Example narrative remapping
    # app.remap_narrative("examples/source_narrative.txt",
    #                     "examples/target_narrative.txt",
    #                     "examples/input_narrative.txt",
    #                     "results/transformed_narrative.txt",
    #                     "results/narrative_comparison.png")

    print("Semantic Drift, Cultural Linguistics, and Narrative Remapping Library")
    print("Usage examples are commented out in the main() function.")
    print("This library provides tools for:")
    print("1. Analyzing semantic drift across time periods")
    print("2. Identifying cultural linguistic patterns in text corpora")
    print("3. Remapping narratives based on structural transformations")


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