SENTENCE SLICING METHODS REPORT

May 19, 2025

Yi-Chun (Rimi) Chen
Data-mining Lab, Yale

Contents

Chapter 1

Sentence Slicing

1.1 Method 1: Sentence Splitting and Annotation Matching

To segment messages into individual sentences, we apply the `sent_tokenize` function from the Natural Language Toolkit (NLTK)[1]. This method is based on the Punkt sentence tokenizer, an unsupervised algorithm that identifies sentence boundaries using statistical models of abbreviations, collocations, and sentence starters.

After sentence segmentation, each sentence is associated with its closest annotation span (if applicable), using fuzzy string matching. The full pipeline consists of the following steps:

1. **Sentence Splitting.** Each message is split into individual sentences using `nltk.sent_tokenize`. This enables finer-grained analysis of message content, especially useful in medical or informal communication.

2. **Sentence ID Assignment.** Each sentence is given a unique `sentence_id`, formed by concatenating the `message_id` and the sentence's index within the message, i.e., `[message_id]_[sentence_index]`.

3. **Annotation Matching.** For each sentence, we compare it against all available annotation spans using normalized Levenshtein similarity, computed via Python's `difflib.SequenceMatcher`. This allows approximate matching in cases where an annotation only partially overlaps with a sentence.

[1] `https://www.nltk.org/api/nltk.tokenize.html`

4. **Threshold Filtering.** The annotation span with the highest similarity score is assigned to the sentence only if its score exceeds a threshold of 0.6. Otherwise, the sentence is considered unmatched. This threshold was chosen to balance between precision and recall: lower thresholds risk false positives, while higher thresholds may miss partial but semantically relevant matches.

Design Considerations.

- Sentence-level segmentation is a natural first step for linguistic processing, as sentences are self-contained semantic units.
- Approximate string matching improves robustness, especially when dealing with annotations that do not exactly align with sentence boundaries.
- The threshold of 0.6 was chosen empirically and can be tuned to better match domain-specific language characteristics.
- Sentences that do not sufficiently match any annotation span are left unlinked to avoid introducing noise.

Output Format. The final structure is a list of message records, each containing a list of sentence entries:

```
{
  "message": "Full message text...",
  "message_id": 0,
  "sentences": [
    {
      "sentence_id": "0_0",
      "sentence": "First sentence.",
      "most_close_annotation_span": "Matched annotation text",
      "subsentences": []
    },
    ...
  ]
}
```

Chapter 2

Sub-sentence Slicing

2.1 Method 2: Subsentence Splitting and Annotation Matching

To further segment each sentence into finer semantic units, we apply clause-level parsing using the spaCy dependency parser[1]. This method is based on syntactic rules for identifying clauses—defined as subject–predicate structures such as independent or dependent clauses—and uses both grammatical roles and punctuation cues to extract clause-like spans.

After clause segmentation, each subsentence (clause) is associated with its closest annotation span (if applicable), using the same fuzzy string matching technique as in Method 1. The full pipeline consists of the following steps:

1. **Clause-Based Subsentence Splitting.** Each sentence is parsed using the spaCy dependency parser. Tokens with dependency labels that commonly signal clause boundaries—such as `ccomp` (clausal complement), `advcl` (adverbial clause), `conj` (coordinate clause), and `parataxis`—as well as punctuation delimiters (e.g., commas and semicolons) are used to segment the sentence into subsentences (clauses). If no boundary is detected, the sentence itself is used as a fallback.

2. **Subsentence ID Assignment.** Each clause is given a unique `subsentence_id`, formed by combining the `message_id`, the sentence index, and the clause index within the sentence, i.e., `[message_id]_[sentence_index]_[clause_index]`.

[1] `https://spacy.io/api/dependencyparser`

3. **Annotation Matching.** For each clause, we compare it against all available annotation spans using normalized Levenshtein similarity, computed via Python's `difflib.SequenceMatcher`. This enables approximate alignment even when the clause only partially overlaps with the annotation span.

4. **Threshold Filtering.** The annotation span with the highest similarity score is assigned to the clause only if its score exceeds a threshold of 0.6. Otherwise, the clause is considered unmatched. The same threshold as in Method 1 is used to ensure consistency across segmentation levels.

Design Considerations.

- Clause-level segmentation captures finer-grained linguistic units than full sentences and allows closer alignment with annotation spans that describe partial actions, conditions, or preferences.
- The use of syntactic dependency labels and punctuation reflects natural clause boundaries and offers robust splitting even in informal or complex sentence structures.
- Using the same similarity threshold as Method 1 ensures consistency while improving annotation coverage for shorter, more focused units.
- Leaving unmatched clauses unlinked avoids introducing ambiguity or noise into downstream classification or reasoning tasks.

Output Format. The final structure is a list of message records, each containing a list of sentences, with each sentence containing a list of subsentence entries:


```
{
  "message": "Full message text...",
  "message_id": 0,
  "sentences": [
    {
      "sentence_id": "0_0",
      "sentence": "Please schedule a follow-up and send the report.",
      "most_close_annotation_span": "send the report",
      "subsentences": [
        {
```

```
          "subsentence_id": "0_0_0",
          "subsentence": "Please schedule a follow-up",
          "most_close_annotation_span": ""
        },
        {
          "subsentence_id": "0_0_1",
          "subsentence": "send the report",
          "most_close_annotation_span": "send the report"
        }
      ]
    },
    ...
  ]
}
```

Chapter 3

Stats

3.0.1 Annotation-to-Structure Alignment Analysis

Table 3.1: Evaluation of Annotation Coverage Across Segmentation Levels

Metric	Count	Match Rate
Total Messages	726	–
Total Sentences	1,874	–
Total Subsentences (Clauses)	4,210	–
Total Annotations	2,602	–
Matched Annotations (Sentence Level)	1,659	63.76%
Matched Annotations (Subsentence Level)	2,024	77.79%

To assess the structural alignment and consistency of the dataset's annotations, we evaluated how well the labeled spans could be matched to sentence-level and clause-level (subsentence) segments. Table 3.1 summarizes the results. Sentence-level segmentation matched approximately 63.76% of the annotation spans, while clause-based subsentence segmentation improved the match rate to 77.79%. This 14% increase suggests that many annotations were made at a finer granularity than full sentences, aligning more naturally with clause-level linguistic units.

These results help gauge the quality of the dataset. The high clause-level match rate indicates that a majority of annotations are well grounded in syntactic structure, which supports their interpretability and potential utility for downstream modeling tasks. In contrast, the remaining 22% of annotations that could not be matched at either level may point to a range of issues, including annotation noise, inconsistency in span boundaries, or expressions that do not map cleanly to standard syntactic segments. This highlights

areas where the annotation process could benefit from refinement, such as reviewing span definition guidelines or introducing context-aware segmentation methods. Overall, segmentation-based matching provides a useful lens for assessing the linguistic coherence and structural quality of annotated text datasets.