12 Preprocessing and Context Representation

12.1 Preprocessing

Regarded naively, an input text feed to an IE system might appear to be flat data without visible structure; just a sequence of characters. This is a wrong impression—there is structure in any text. At a low level, text can be considered as a sequence of tokens (words, numbers, punctuation). In natural language texts, tokens are arranged in sentences. Several sentences are grouped in paragraphs, which are grouped in sections (which in turn might be grouped in higher-order sections). In structured text formats the higher-level structure (usually down to the paragraph level) is explicitly coded, but the lower-level structure (sentences; sentence constituents such as verb groups or noun phrases; tokens) must usually be induced.

As explained in Section 9.2, we have decided to utilize XML as a generic input format that allows expressing any structural information about a text, as long as the resulting structure is a tree. (Overlapping markup is not possible since it would result in the text structure being a general graph instead of a tree.)

Any XML-annotated documents can be handled without requiring conversion—our algorithms are agnostic about the semantics of the used tag set, since they only use structural information, leaving it to the classifier to learn and recognize any implied semantics that are relevant for classification. We also accept HTML input, using *JTidy* [JTi] to clean up any markup errors and inconsistencies and to ensure a valid XML structure.

In case of plain text input (such as the *Seminar Announcements* and *Corporate Acquisitions* corpora used for evaluation, cf. Chap. 17), we invoke the *txt2html* [txt] converter to convert the input into XHTML. This converter relies to heuristics to recognize and explicitly represent (in HTML tags) structural and formatting information (blocks of emphasized text, lists, headers etc.) that is frequently implicit in plain (ASCII) texts and that would be lost if we just processed the text as an unstructured series of tokens.

Any other document formats can be processed by integrating a suitable converter into the system or by converting them to XML or HTML prior to processing.

In a further preprocessing step, the text is augmented with explicit linguistic information. We use the well-known *TreeTagger* [Tre] to:

- Divide a text into sentences.[1] TreeTagger has not been originally designed as a sentence splitter—it does not mark the beginning and ending of sentence, but only the ending (by POS-tagging punctuation characters such as '.', '?', '!' as end-of-sentence marker if they are used that way). Normally, sentences can be

[1] By adding `sent` (sentence) elements to the XML tree.

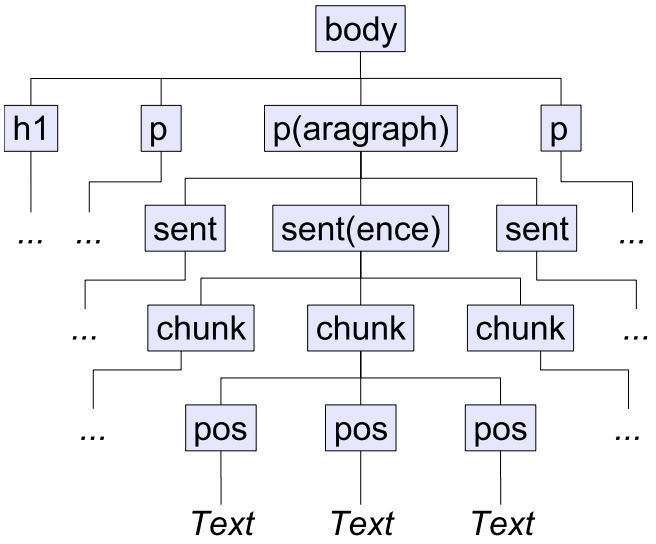

Figure 12.1: Partial DOM Tree of a Simple HTML Document with Linguistic Annotations

assumed to extend from one end-of-sentence marker (or from the begin of the text in case of the very first sentence) to the next one. However, sometimes there are intervening text fragments that are not part of any full sentence, e.g. section headings. This makes the placement of the begin-of-sentence marker somewhat less straightforward—we use the XML repair algorithm that will be described in the next chapter to insert the begin-of-sentence tags in the most likely position.

- Split sentences into "chunks" such as verb groups, noun phrases and prepositional phrases.[2]
- Tokenize the input into a sequence of *parts-of-speech* (words, numbers and punctuation) and determine their syntactic categories and normalized base forms.[3]

For German texts, we additionally use *SPPC*[4] to split compounds into segments.[5]

The output of the preprocessing tools is converted to the XML markup mentioned in the footnotes and merged with the explicit markup of the source document. The schema describing the linguistic markup is given in Appendix A; the merging algorithm will be described in the following chapter. After preprocessing, a text is represented as a DOM (Document Object Model) tree. The structure of the DOM tree for a simple HTML document (containing a section heading and several paragraphs) is shown in Fig. 12.1.

Figure 12.2 shows a preprocessed file from the *Seminar Announcements* corpus—the same file that was used as example in Sec. 3.4 and in Chap. 9. XML elements

[2] By adding `const` (sentence constituent) elements with a `type` attribute that identifies the type of the constituent.
[3] By adding `pos` (part-of-speech) elements with `type` and `normal` attributes.
[4] A successor of the *SMES* system described in [Neu02].
[5] The list of segments is stored in the `segments` attribute of `pos` tags that are compounds; the base forms of all segments are stored in the `normalSegments` attribute and the `baseSegment` attribute contains the base form of the main segment.

```
− <html>
  + <head></head>
  − <body>
    + <dl></dl>
    + <p></p>
    − <p>
      + <sent></sent>
      − <sent>
        − <const type="NC">
            <pos type="DT" normal="the">The</pos>
            <pos type="NN" normal="lecture">lecture</pos>
          </const>
        − <const type="VC">
            <pos type="VVZ" normal="begin">begins</pos>
          </const>
        − <const type="PC">
            <pos type="IN" normal="at">at</pos>
          − <const type="NC">
              <pos type="CD" normal="@card@">3:30</pos>
              <pos type="RB" normal="p.m">p.m</pos>
              <pos type="VVN" normal="follow">followed</pos>
            </const>
          </const>
        − <const type="PC">
            <pos type="IN" normal="by">by</pos>
          − <const type="NC">
              <pos type="DT" normal="a">a</pos>
              <pos type="NN" normal="reception">reception</pos>
            </const>
          </const>
        − <const type="PC">
            <pos type="IN" normal="in">in</pos>
          − <const type="NC">
              <pos type="NP">Hamerschlag</pos>
              <pos type="NP" normal="Hall">Hall</pos>
            </const>
          </const>
          <pos type="," normal=",">,</pos>
        − <const type="NC">
            <pos type="NP" normal="Room">Room</pos>
            <pos type="CD" normal="@card@">1112</pos>
          </const>
          <pos type="SENT" normal=".">.</pos>
        </sent>
      + <sent></sent>
      + <sent></sent>
      </p>
    </body>
  </html>
```

Figure 12.2: Processed File from the Seminar Announcements Corpus

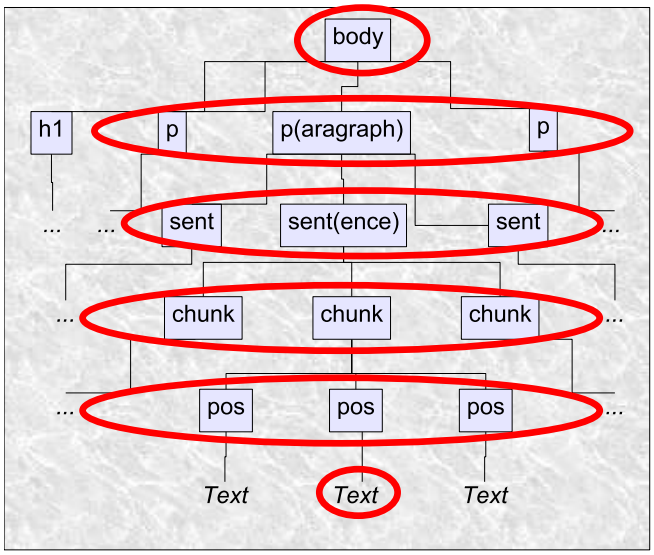

Figure 12.3: Inverted Subtree of the Elements Considered for a Context Representation

marked with '+' have been collapsed to keep the sample reasonably short.

12.2 Tree-based Context Representation

Typically, the *context window* considered by IE algorithms comprises either the nearest tokens/words (e.g. [Cir01]) or some predefined syntactic elements of the current sentence (e.g. [Sod95]). The hierarchical tree structure obtained by our preprocessing approach allows a more flexible context model: the context of a node contains the nearest nodes around it. The context we consider for each token includes features about:

- The token itself and the POS (part-of-speech) element it is in.
- Up to four preceding and four following siblings[6] of the POS element (neighboring parts-of-speech, but only those within the same sentence chunk).
- Up to four ancestors of the element (typically the embedding chunk, sentence, paragraph or related unit, etc.)
- Preceding and following siblings of each included ancestor—the number of included siblings is decremented for each higher level of ancestors (three for the direct parent, i.e. three preceding and three following chunks; two for the "grandparent", i.e. sentence; etc.)

The information included in our context representations thus consists in a subtree of the full DOM tree, starting from a leaf node (containing the token to classify) and extending from there to the four nearest ancestors and their siblings. Context representations are thus based on *inverted subtrees* of the whole document tree. Figure 12.3

[6] We use the terms *preceding sibling*, *following sibling*, *parent*, and *ancestor* as defined by the XPath standard [XPa].

continues the example from Fig. 12.1, highlighting the elements to be included in the context representation for the leaf node at the bottom.

In addition to this DOM tree–based context, we add information on the last four attribute values found in the current document, similar to the *lastTarget* variable used in [Pes03]. This allows the classifier to learn about positional relations among attributes to extract (for example, the END TIME of a seminar announcement will usually follow the START TIME, and the LOCATION of the seminar will often follow both).

In the application phase, this information about preceding attribute values will be somewhat noisy since the true attribute values (answer keys) are not known, and the predicted attribute values might be erroneous; on the other hand, other context information such as linguistic annotations will also contain occasional noise and errors. As part of the evaluation phase of our work, we will perform an ablation study to investigate the influence of the various source of information that form the context representations, determining whether and how useful they actually are (in spite of noise).

In the DOM tree creating during preprocessing, all leaf nodes are POS elements, each representing a single word or another part-of-speech. Each POS element contains a text fragment for which we include several features:

- The text fragment, both in original capitalization and converted to lower-case;
- Prefixes and suffixes from length 1 to 4, converted to lower-case;[7]
- The length of the fragment;[8]
- The word shape the fragment (one of *lowercase, capitalized, all-caps, digits, digits+dots, digits+colons, alphanumerical, punctuation, mixed* etc.)

Additionally, the semantic class(es) the fragment belongs to are listed, if any. For this purpose, a configurable list of dictionaries and gazetteers are checked. Currently we use the following semantic sources:

- An English dictionary;[9]
- Name lists from US census;[10]
- Address suffix identifiers from US Postal Service;[11]
- A small list of titles from Wikipedia.[12]

Since all textual content of preprocessed documents is wrapped inside POS elements, all other elements are inner nodes which contain child elements instead of directly containing text. For *chunk* elements, we include the normalized form of the right-most POS that is not part of a sub-chunk as head word.[13] For elements containing

[7] Prefixes and suffixes that would contain the whole fragment are omitted.
[8] Both the exact value and the rounded square root as a less sparse representation.
[9] `http://packages.debian.org/testing/text/wamerican`
[10] `http://www.census.gov/genealogy/names/`
[11] `http://www.usps.com/ncsc/lookups/abbreviations.html`
[12] `http://en.wikipedia.org/wiki/Title`
[13] This is a somewhat language-specific heuristic aimed at languages such as English and German, where articles and adjectives are usually placed to the left of a noun. It should be modified for languages such as Spanish, where attributes frequently follow the noun they modify.

chunks (such as *sentence*), the head words of the left-most and the right-most chunk are included.

For other elements, the name of the element is added to the feature set. Any XML attribute name/value pairs of the included elements are also added.[14] The position of each element relative to the token to represent is encoded in the generated features; for the represented POS element and its ancestors, we also store the position of the element within its parent.

This results in a fairly high number of features representing the context of each token. The features are arranged in an ordered list to allow recombination via feature combination techniques such as OSB (cf. Sec. 11.2); the resulting feature vector is provided as input to the classifier.

12.3 Tokenization

Since we model information extraction as a token classification task (cf. Chap. 10), tokenization is especially important for our approach. Only complete tokens can be extracted—an attribute value can comprise one or multiple whole tokens, but it cannot comprise partial tokens. Hence, attribute values whose borders do not correspond to token borders are impossible to extract correctly. Tokenization should be precise enough to avoid introducing such inevitable errors.

On the other hand, increasing the number of tokens by choosing a more specific tokenization schema increases both the risk of errors and the runtime of the algorithm (which with our default classifier Winnow+OSB is linear to the number of tokens, cf. Chap. 11). Therefore we should not increase the number of tokens unnecessarily.

An initial tokenization is already performed during linguistic preprocessing. As explained above (Sec. 12.1), the linguistic preprocessor will split a text into a series of part-of-speech (POS) tokens. This takes care of most tokenization issues, including the separation of most punctuation symbols from preceding words. However, some of the expressions regarded as single POS elements by the preprocessor are not yet sufficiently granular for information extraction. For example, the expression "12:00-1:30" is considered a single POS element by TreeTagger, but it contains both the START TIME (**"12:00"**) and the END TIME (**"1:30"**) of a seminar announcement (cf. Sec. 17.1). To be able to extract (or to train) either attribute value, we need a more fine-grained tokenization.

For this purpose we use a regular expression–based tokenizer that uses a configurable list of regular expression patterns for tokenization. By default, four patterns are used, one for matching alphanumerical sequences (normal words and numbers), one for matching monetary amounts, another one for punctuation characters (commas, dots, quotation marks etc.) and the last one for any other printable characters (such as mathematical symbols). The exact patterns used are given in Table 12.1—they are chosen in a way that allows exactly matching and extracting all attribute values that

[14] XML attributes specify the *type* of parts-of-speech and chunks as well as the *normalized form* of parts-of-speech, as stated above (Sec. 12.1); other XML attributes might be present in the original or converted XML representation of a document.

Alphanumeric, can contain one of ".,:" between digits: `(?:\p{N}[.\,:]\p{N}|[\p{L}\p{M}\p{N}])+`
Currency symbols, can be followed by numbers (incl. inner ".,;"): `\p{Sc}+(?:\p{N}[.\,:]\p{N}|\p{N})*`
Single punctuation sign, possibly repeated: `(\p{P})\1*`
Other Symbols (non-currency): `[\p{Sm}\p{Sk}\p{So}]+`

Table 12.1: Regular Expressions Used for Tokenization

do occur in the evaluation corpora (Sec. 17.1) and all that are likely to occur in other corpora,[15] but that does not generate more tokens that necessary for this purpose.

The first pattern matches most tokens—it allows any alphanumeric sequences, i.e., sequences containing letters and digits but no other characters. Additionally, it allows the punctuation characters `.,:` (dot, comma, colon), but only if they are surrounded by digits. These characters are typical within numeric expressions such as "1.25" or "1,000,000" and time expressions such as "12:00pm" and should not cause such expressions to be split.

The second pattern matches currency symbols and currency amounts, such as "$9.99" or "€50,000". The third one matches a single punctuation character or several repetitions of the same punctuation character (e.g. "..."). It will not match sequences of different punctuation characters (which are quite rare, anyway) as a single token, since some token borders might occur between different punctuation characters. For example, in the sentence "The speaker, Guy L. Steele, Jr., will talk about..." the SPEAKER attribute value to extract is **"Guy L. Steele, Jr."**, including the dot after "Jr" but excluding the subsequent comma.

The fourth and last pattern matches sequences of any other symbols, e.g. mathematical symbols that might occur in scientific texts.

[15] So far, our system has been used for two or three additional corpora containing German or English texts, and the tokenization turned out to be appropriate for all of them without needing adjustments.