23-06-2023

**Chapter 3: Processing Raw Text (Part 2)**

1. **Normalising Text:**
2. **Stemmers:**

NLTK includes several off-the-shelf stemmers. NLTK’s stemmers handle a wide range of irregular cases. The Porter and Lancaster stemmers follow their own rules for stripping affixes.

Stemming is not a well-defined process, and we typically pick the stemmer that best suits the application we have in mind. The Porter Stemmer is a good choice if you are indexing some texts and want to support search using alternative forms of words.

1. **Lemmatization:**

The WordNet lemmatizer removes affixes only if the resulting word is in its dictionary. This additional checking process makes the lemmatizer slower than the stemmers just mentioned. The WordNet lemmatizer is a good choice if you want to compile the vocabulary of some texts and want a list of valid lemmas (or lexicon headwords)

Another normalization task involves identifying non-standard words, including numbers, abbreviations, and dates, and mapping any such tokens to a special vocabulary. For example, every decimal number could be mapped to a single token 0.0, and every acronym could be mapped to AAA. This keeps the vocabulary small and improves the accuracy of many language modeling tasks.

1. **Regular Expressions for Tokenising text:**

Tokenization is the task of cutting a string into identifiable linguistic units that constitute a piece of language data. Although it is a fundamental task, we have been able to delay it until now because many corpora are already tokenized, and because NLTK includes some tokenizers.

1. **Simple approaches to Tokenisation:**

The very simplest method for tokenizing text is to split on whitespace.

We could split this raw text on whitespace using raw. split (). To do the same using a regular expression, it is not enough to match any space characters in the string, since this results in tokens that contain a \n newline character; instead, we need to match any number of spaces, tabs, or newlines.

1. **NLTK’s Regular Expression Tokenizer:**

The function nltk. regexp\_tokenize () is similar to re. findall ().

. However, nltk. regexp\_tokenize () is more efficient for this task, and avoids the need for special treatment of parentheses. For readability we break up the regular expression over several lines and add a comment about each line. The special (?x) “verbose flag” tells Python to strip out the embedded whitespace and comments.

1. **Further Issues with Tokenisation:**

Tokenization is a challenging task that requires careful consideration. There is no one-size-fits-all solution, and the definition of a token may vary depending on the specific application domain. To develop an effective tokenizer, it is beneficial to have access to manually tokenized raw text, which can serve as a reliable reference for evaluating the output of your tokenizer.

The NLTK corpus collection provides a sample of Penn Treebank data, which includes both the raw Wall Street Journal text and its tokenized version. Comparing your tokenizer's results with the high-quality tokens from this "gold-standard" dataset can be helpful in assessing its performance.

Another aspect to consider during tokenization is the handling of contractions like "didn't." In sentence analysis, it is often more advantageous to normalize such contractions into separate forms, such as "did" and "n't" (or "not"). This normalization can be achieved using a lookup table or a similar mechanism.

1. **Segmentation:**

Tokenization is an instance of a more general problem of segmentation.

1. **Sentence Segmentation:**

Manipulating texts at the level of individual words often presupposes the ability to divide a text into individual sentences.

In other cases, the text is available only as a stream of characters. Before tokenizing the text into words, we need to segment it into sentences. NLTK facilitates this by including the Punkt sentence segmenter (Kiss & Strunk, 2006).

Sentence segmentation is difficult because a period is used to mark abbreviations, and some periods simultaneously mark an abbreviation and terminate a sentence, as often happens with acronyms like U.S.A.

1. **Word Segmentation:**

Tokenizing text can be more challenging in writing systems where word boundaries are not visually represented. For instance, in Chinese, the absence of explicit word boundaries poses difficulties for tokenization. A three-character string like 爱国人 (ai4 "love" [verb], guo3 "country," ren2 "person") can be tokenized as 爱国/人, meaning "country-loving person," or as 爱/国人, meaning "love country-person."

Similar challenges arise in processing spoken language, where the listener needs to segment a continuous stream of speech into individual words. This becomes particularly complex when word boundaries are unknown, as is the case for language learners, such as children hearing utterances from their parents.

To address this, a representation method is needed to separate text content from segmentation. An approach is to annotate each character with a boolean value indicating the presence or absence of a word-break after it. This annotation facilitates the segmentation process and can be used for tasks like "chunking" in natural language processing.

The provided example demonstrates the process of reconstructing segmented text from its representation using strings of zeros and ones. The segmentation strings, such as seg1 and seg2, indicate the presence of word boundaries. By applying the segment() function, the original segmented text can be obtained.

The task of segmentation becomes a search problem, where the goal is to find the bit string that correctly segments the text into words. An objective function is defined to measure the quality of segmentation based on the lexicon's size and the information required to reconstruct the source text from the lexicon.

The cost of storing the lexicon and reconstructing the source text can be computed using the evaluate() function. By comparing different segmentations, it is possible to determine the optimal segmentation with the lowest cost.

To search for the best segmentation pattern, a non-deterministic search method called simulated annealing is employed. This method involves randomly perturbing the boundaries between segments and gradually reducing the perturbation as the "temperature" decreases. The goal is to find the segmentation with the lowest cost based on the objective function.

With sufficient data, automatic segmentation of text into words can be achieved with reasonable accuracy. This approach is particularly useful for tokenizing writing systems that lack visual cues for word boundaries.

1. **Formatting from Lists to Strings:**

Often, we write a program to report a single data item, such as a particular element in a corpus that meets some complicated criterion, or a single summary statistic such as a word-count or the performance of a tagger. More often, we write a program to produce a structured result; for example, a tabulation of numbers or linguistic forms, or a reformatting of the original data. When the results to be presented are linguistic, textual output is usually the most natural choice. However, when the results are numerical, it may be preferable to produce graphical output. In this section, you will learn about a variety of ways to present program output.

Concepts under this section such as:

* From lists to strings
* String and Formats
* Lining things up
* Writing results to a file
* Text wrapping

Are executed practically in the notebook.