Caleb Thomas
calebt4
CS 410

NLTK for Information Extraction

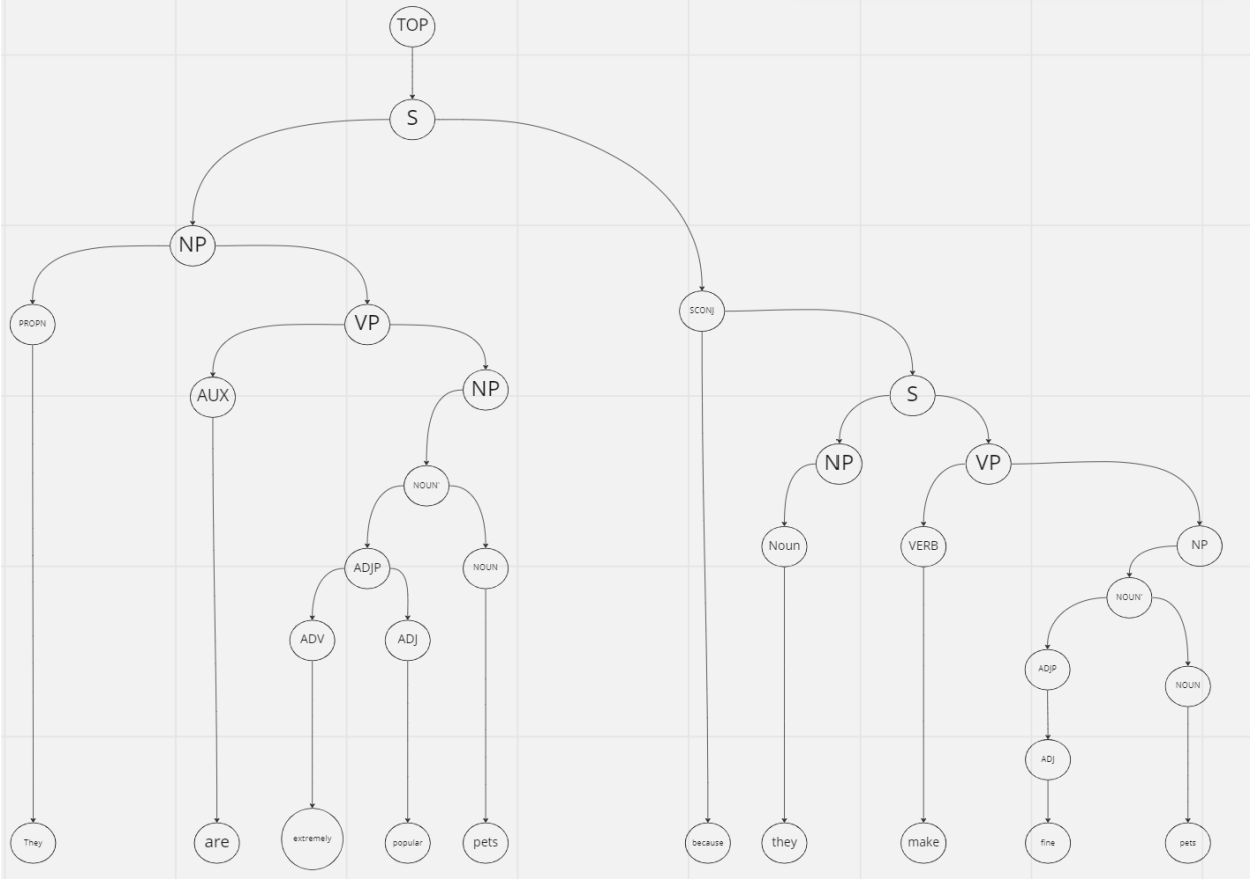
NLTK is an expansive Python library that provides various ways to analyze and clean text data. A few examples are lemmatizing, Chunking, and parts of speech tagging. An important task related to text retrieval, clustering, and classification is Information Extraction. NLTK offers an architecture specifically for this task.

The first part of NLTK architecture for Information Extraction is sentence segmentation (Bird, Klein, Loper). The default sentence tokenizer that NLTK uses is the PunktSentenceTokenizer. This tokenizer uses an unsupervised algorithm to create a set of parameters from the given corpus that it can use to find where a sentence boundaries. Part of this unsupervised algorithm is bigrams. Bigrams are used for tracking what words appear next to periods, and then if the frequency of a bigram is high enough, it will be used to help detect sentence boundaries. After this stage is complete, the next in line is word tokenization.

The word tokenization step follows a similar structure. The word tokenizer is invoked by calling nltk.word_tokenize, and from here, NLTK uses its default word tokenizer, the TreebankWordTokenizer. This class performs a few different tasks using regular expressions. First, it splits contractions, then it splits off commas and single quotes, and finally separates periods that appear at the end of a sentence. With sentence segmentation and word tokenization complete, each token in the corpus can now be tagged with its corresponding part of speech tag.

Like with sentence segmentation and word tokenizer, the NLTK part of speech tagger can be called with a similar convention of nltk.pos_tag. The default part of speech tagger is an AveragedPerceptron. The way it works is that the Perceptron is trained on a tag-labeled corpus. Once training is complete, the weights are averaged, and now inferences can be made on an

unlabeled corpus to tag it. By combining the Perceptron with a technique called Chunking, which works by grouping similar parts of a sentence together and then stepping into each group, getting more granular with each step, NLTK is able to efficiently tag a corpus in a detailed manner. This process can look similar to a binary tree. Below is an example I created using Miro and the sentence, "They are extremely popular pets because they make fine pets."



The first part of the sentence, "they are extremely popular pets," is grouped under an NP or Noun Phrase, and then further down that branch of the tree gets more detailed until the tree's leaf is reached, which contains a single word. With entity detection done, the final step of the process can be completed, which is relation detection.

With the given corpus tagged, entity detection can now be done. To do entity recognition, NLTK once again uses Chunking. NLTK, by default, uses a multiclass classifier which can be

used for classifying people, organizations, etc. (Bird, Klein, Loper). Now that entity recognition is complete, the final step of relation detection can be done.

Relation detection requires all of the previous steps to be completed. One of the ways NLTK accomplishes relation detection is by looking at the text between two entities (Bird, Klein, Loper).

Overall the NLTK library has many useful features and excels at the information extraction task.

Citations

Bird, Steven, Ewan Klien, and Edward Loper. “Natural Language Processing with Python.” nltk.org, September 4, 2019. https://www.nltk.org/book/ch07.html.

Bird, Steven, “NLTK Github” https://github.com/nltk/nltk