CS 6320.002: Natural Language Processing
Fall 2019

Homework 4 – 92 points
Issued 07 Oct. 2019
Due 8:30am 21 Oct. 2019

Deliverables: A tarball or zip file containing your code and your PDF writeup. Note that in this assignment, you have been given some skeleton code. Do not change anything in the skeleton code except where these instructions and the code comments tell you to do so! If you want to test your code as you go along, you can put your tests in `main()`, just make sure you comment them out before turning in.

0 Getting Started

Make sure you have downloaded the following files:

- `ie.py`, the skeleton code that you will be filling in
- `wikipedia_sentences.txt`, the training data
- `test.txt`, the test data

Make sure you have installed NLTK, `https://www.nltk.org/`.

In this assignment, we are going to extract the hyper/hyponym relation using Hearst patterns. Recall that Hearst patterns capture the hyper/hyponym relation among noun phrases that occur in five specific contexts:

NP {, NP}* {,} (and|or) other NP_H	temples, treasuries, and other important civic buildings
NP_H such as {NP,}* {(or|and)} NP	red algae such as Gelidium
such NP_H as {NP,}* {(or|and)} NP	such authors as Herrick, Goldsmith, and Shakespeare
NP_H {,} including {NP,}* {(or|and)} NP	common-law countries, including Canada and England
NP_H {,} especially {NP}* {(or|and)} NP	European countries, especially France, England, and Spain

1 Loading the Data – 12 points

We will be extracting relations from Wikipedia sentences. The format of the Wikipedia corpus is two sentences per line, separated by a tab. The second sentence is a lemmatized verison of the first. Both sentences are already tokenized (so we can get the tokens simply by calling `split()`).

Fill in the function `load_corpus(path)`. The argument `path` is a string. The function should do the following:

- Open the file at `path`.
- Use a list comprehension* to convert the lines of the file into a list of (`sentence`, `lemmatized`) tuples.

  - In each tuple, `sentence` should be a list of tokens; `lemmatized` should be a list of lemmatized tokens.
  - Make sure you use `strip()` as needed to remove any extra whitespace.
- Return the list.

*Note that you don't have to use a single list comprehension to do all the work. It can be done, but it is also okay to split the list comprehension step into several smaller steps, as long as the final result is the same.

For testing, we are using the BLESS dataset. The format of this data is three words per line, separated by tabs. The first word is a candidate hyponym, the second word is a candidate hypernym, and the last word is either "True" or "False" (ie. are the two candidates really in a hyper/hyponym relation).

Fill in the function `load_test(path).` The argument `path` is a string. The function should do the following:

- Open the file at `path`.
- Convert the lines of the file into two sets of (hyponym, hypernym) tuples.
  - One set should contain all pairs with the label "True"; the other set should contain all pairs with the label "False".
  - The hyponyms and hypernyms in the tuples should be strings.
  - Again, make sure you use `strip()` as needed to remove any extra whitespace.
- Return a tuple (true set, false set).

2 NP Chunking – 36 points

To use Hearst patterns to extract relations from the Wikipedia corpus, we need to be able to identify the noun phrases in a sentence. Since we only care about noun phrases (and not any other phrases or relationships between phrases), we don't need to do a full parse; we can do shallow parsing, aka. *chunking*. We will develop a simple, rule-based chunker to identify noun phrases in a POS-tagged sentence.

We are using the `nltk.RegexpParser` class, which chunks input based on a grammar consisting of regular expressions. The grammar is a string with the format `NP: {pattern}`. Yes, the curly braces are required. `pattern` is a regular expression for what POS sequences (not word sequences!) are considered an NP.

The formatting of this pattern is a little different than normal regular expressions. We use the angle brackets `<>` to indicate which parts of the pattern belong to which POS tag in the chunk. For example, if the pattern is `<ab><cd><ef>`, then `ab` is the pattern for the first POS tag in the chunk, `cd` is the pattern for the second tag, and `ef` is the pattern for the third. The optional, count, and wildcard operators can still be used both inside and outside of the angle brackets. For example, `<ab.?><cd>+` would match one tag matched by `ab.?`, followed by zero or one tags matched by `bc`.

Fill in the global variable `NP_grammar` with a pattern for matching NPs. The pattern should match this POS tag sequence: optional determiner, zero or more adjectives, and one or more nouns (in that order). We are using the Penn Treebank POS tagset this time:
`https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

Now for the actual chunking.

Fill in the function `chunk_lemmatized_sentence(sentence, lemmatized, parser).` The arguments `sentence` and `lemmatized` are lists of tokens, and `parser` is an instantiated `nltk.RegexpParser` using the grammar we just filled in. The function should do the following:

- Use `nltk.pos_tag()` to convert `sentence` into a list of (token, tag) tuples.
- Use the tagged `sentence` to convert `lemmatized` into a list of (token, tag) tuples as well.*
- Use `parser.parse()` to convert `lemmatized` into an `nltk.Tree`.
- Use the helper function (which we will fill in next) `tree_to_str()` to convert the Tree into a list of chunks.
- Use the helper function (which we will fill in next) `merge_chunks()` to convert the list of chunks into a string.
- Return the string.

*Why do we need this step? Ultimately we want to extract relations from the lemmatized version of the Wikipedia corpus, since, if we look at the test data, we find that the candidate hyponyms and hypernyms are lemmatized. If we don't also lemmatize our extracted relations, we won't be able to match the test data. So we will have to do POS tagging on the original sentence and then assign those tags to the lemmatized version. Luckily, this is not hard to do, since every word in the original sentence corresponds to exactly one word in the lemmatized version.

Writeup Question 2.1: Why do we have to run POS tagging on the original sentence instead of running it directly on the lemmatized sentence? Explain why we don't want to do POS tagging using the lemmatized sentence.

The output of `parser.parse()` is an `nltk.Tree`, so the goal of the helper functions `tree_to_chunks()` and `merge_chunks()` is to convert the `Tree` back into a string so that we can use Hearst patterns to match it. A `Tree` represents an internal node in a tree, and it is basically just a list, where the elements of the list are the node's children. A child could be another `Tree`, or it could be some other object (a leaf node). The documentation for `Tree` is here: `https://www.nltk.org/api/nltk.html#nltk.tree.Tree`

In our case, `parser.parse()` returned the root of the shallow parse tree: any child of the root that is a `Tree` represents an NP chunk, and its children are all of the (token, tag) tuples in the chunk; any child of the root that is not a `Tree` is simply one of the non-NP (token, tag) tuples from the rest of the sentence.

Fill in the helper function `tree_to_chunks(tree).` The argument `tree` is an `Tree`. The function should do the following:

- Initialize an empty list to hold the chunks we will return.
- Iterate through the children of `tree`.
  - Use `isinstance()` to check if a child is an `nltk.Tree`.
  - If it's not, then it is a non-NP (token, tag) tuple; simply append the token to the chunk list.
  - If it is, then each of its children is a (token, tag) tuple that belongs to an NP.
    - Use a list comphrehension to get a list of all the tokens in the NP.
    - Use `join()` convert the list into a single string, with the tokens separated by underscores (eg. `the_hungry_dog`).
    - Tag the string with `NP_` (eg. `NP_the_hungry_dog`).
    - Append the string to the chunk list.
- Return the list of chunks.

The second helper function does two things: if there are multiple NP chunks right next to each other (ie. with no tokens separating them), it merges them into a single large NP chunk; then it merges all of the chunks into a single string.

Fill in the helper function `merge_chunks(chunks).` The argument `chunks` is a list of strings. The function should do the following:

- Initialize an empty list as a buffer to hold pieces of the reconstructed string.
- Iterate through the strings in `chunks`.
  - If the last chunk in the buffer* has the `NP_` tag and the next chunk in `chunks` also has the `NP_` tag, combine them into a single chunk and replace the last chunk in the buffer with this new large chunk (eg. `NP_the_morning NP_flight` becomes `NP_the_morning_flight`).
  - Otherwise, simply append the next chunk in `chunks` to the the buffer.
- Use `join()` to convert the buffer into a single string, with the tokens separated by a single space.
- Return the string.

*Make sure you handle the corner case where the buffer is empty.

Now to actually chunk the Wikipedia corpus! In `main()`, **complete in the line** `wikipedia_corpus = ['Your list comprehension here']` by filling in a list comprehension that uses `chunk_lemmatized_sentence()` to convert the Wikipedia corpus from a list of (sentence, lemmatized) tuples into a list of chunked and tagged strings.

You can test your code so far by checking the first sentence in the Wikipedia corpus. It should be converted into this: "because of NP_fuel_shortage and lack of NP_spare_part , NP_sortie by NP_South_Vietnamese_helicopter and NP_cargo_aircraft shrank by 50 to 70 NP_percent ." Note that there are errors in the chunking here, some due to incorrect POS tagging (eg. "lack"), and some due to deficiencies in our NP chunking pattern (eg. "50 to 70 percent").

3 Hearst Patterns – 12 points

Now let's write some regular expressions to capture the Hearst patterns.

Normally we would use capturing groups to extract which NP in a Hearst pattern match is the hypernym, and which are the hyponyms. Unfortunately, we don't know how many hyponyms there could be in a single sentence, so it's hard to match them with a single regex using groups. Luckily, there's an easier way: all we have to do is match the entire Hearst pattern. Then, because all the NPs are tagged with `NP_`, we can easily tell what the extracted hypernyms and hyponyms are. We can also easily tell which one is the hypernym and which ones are the hyponyms: either the hypernym comes before all of the hyponyms, or it comes after.

Fill in the global variable `hearst`. `hearst` is a list containing five tuples, one for each of the five Hearst patterns on the first page of this assignment. Each tuple consists of a regex for the pattern and a single word: either "before" or "after", which tells you whether the hypernym (NP_H) comes before or after the hyponyms. As you're writing the regexes, remember that the sentences are tokenized (so there is a space between every token, including punctuation like commas and periods) and NPs have been tagged as in Part 2.

Pattern 2 (NP_H, such as NP_1, NP_2, ...) has been filled in for you as an example.

To test your code for this section, you can try to match the examples given in the Hearst pattern table on the first page of this assignment. Just make sure you tokenize them (manually or using `nltk.word_tokenize()`) and tag the NPs first.

Writeup Question 3.1: What are some other patterns that capture the hyponym-hypernym relation? Give at least two other patterns and an example of each. You can look at the training sentences or any other English text to get ideas. You don't have to implement the patterns, just state what they are and give an example sentence demonstrating how each one works.

4 Extraction – 12 points

Finally it's time to do the actual relation extraction. We are going to try to match the Hearst patterns against the sentences in the Wikipedia corpus. If we find a match, we can extract from it one (or more) hypernym-hyponym pairs.

Fill in the generator function `extract_relations(chunked_sentence).` The argument `chunked_sentence` is a string that has been chunked and NP-tagged, as described in Part 2. The function should do the following:

- Iterate through the five regexes in the global variable `hearst`.
  - For each regex, use `re.search()` to look for a match.
  - If a match is found, use `group(0)` to get the full text of the match.
  - Use `split()` to tokenize the match; this will produce a list of strings.

- Use a list comprehension to remove all tokens from the list that do not have the `NP_` tag.
- Use the helper function (which we will fill in next) `postprocess_NPs()` to remove the `NP_` tag and underscores from the NPs in the list.
- Check whether the current `hearst` regex is a "before" rule or an "after" rule.
  - If it's "before", the hypernym is the first token in the list, and the rest are hyponyms.
  - If it's "after", the hypernym is the last token in the list, and the rest are hyponyms.
- Iterate through all the hyponyms.
  - For each hyponym, yield the tuple (hyponym, hypernym).

The helper function gets rid of the `NP_` tag and the underscores that we added back in Part 2. **Fill in the helper function** `postprocess_NPs(NPs).` The argument `NPs` is a list of strings that are the extracted NPs. The function should do the following:

- Use `replace()` to remove the `NP_` tag.
- Use `replace()` to replace the underscores with a single space.
- Return the edited string.

You can again test your code for this section using the examples on the first page of this assignment. For example, the sentence "such authors as Herrick, Goldsmith, and Shakespeare" should produce three tuples: (authors, Herrick), (authors, Goldsmith), and (authors, Shakespeare). Again, make sure you tokenize the sentences and tag the NPs first, or it won't work.

5 Evaluation – 15 points

The last thing to do is evaluate the performance of our rule-based relation extractor. Recall that in Part 1 we created two sets of gold standard (hyponym, hypernym) tuples, one containing all the true relations and one containing all the false relations in the test set. Now let's compare our extracted relations with the gold standard labels and calculate the precision, recall, and f-measure.

Fill in the function `evaluate_extractions(extractions, gold_true, gold_false).` The argument `extractions` is a set of extracted (hyponym, hypernym) tuples, and `gold_true` and `gold_false` are the two gold standard sets from Part 1. The function should return a tuple of (precision, recall, f-measure).

These three metrics are calculated just like in HW2, with two differences. First, instead of checking if the predicted label and the gold standard label are the same, we are checking which of the two gold standard sets an extracted tuple is in. Second, it's possible that we extracted a tuple that isn't in either gold standard set; we don't know if the extraction is correct or not. In such a case, we want to skip that pair in our evaluation because we don't know what it really is.

All the functions are now filled in, so we can run the script, which will use your filled-in functions to load the data, chunk and tag it, extract relation pairs, and evaluate. It will print out the precision, recall, and f-measure of your extractions.

Writeup Question 5.1: What precision, recall, and f-measure did you get?

Writeup Question 5.2: Look at the test data and think about the hyponyms and hypernyms that our system extracts. Do you notice anything that might be hurting the performance of our system? How would you fix it? (You don't have to actually implement it, just state what you would do.)

6 Meta Questions – 2 points

Writeup Question 6.1: How long did this assignment take you to complete (not counting extra credit)?

Writeup Question 6.2: Did you discuss this homework with anyone? If so, who?

7 Extra Credit – 8 points

Remember those two extra patterns you came up with in Part 3? Let's think up a couple more and implement them!

Add at least four new patterns to the global variable `hearst`. You can use the two from Part 3. Again, you can skim the Wikipedia data or any other English text to get ideas for other hyponym-hypernym patterns.

Writeup Question 7.1: Explain the patterns you came up with and give an example sentence for each.

Now rerun the script, and let's see if the extra patterns improved our performance!

Writeup Question 7.2: What are the new precision, recall, and f-measure?