CIS 430/530 Fall 2012 HW 4

Instructor: Ani Nenkova
TA: Kai Hong, Jessy Li, Achal Shah

Released: November 21, 2012
Due: 11:59PM November 30, 2012

Overview

This assignment focuses on dependency parsing and WordNet, there will also be a part on visualization.

Deliverables

You will submit the code for the functions you are asked to implement, along with some comments in the code that are asked by some of the questions. There are in total 100 points for the two homework problems.

Submitting your work

The code for your assignment should be placed in a single file called `hw4_code_yourpennkey.py` where `yourpennkey` is your Penn Key. Since my (Jessy) pennkey is "ljunyi", I would submit the following files: `hw4_code_ljunyi.py`. You should also submit your visualization png files, which should be called "nouns.png" and "verbs.png", as well as the alternative texts in problem 2, named "1612890.alt", "536101.alt", "88246.alt"

To electronically submit homework, if you are not already working on Eniac, you need to place the file containing your solution on your SEAS account storage. One way to do this would be to use an SFTP client such as FileZilla or WinSCP.

Then connect via ssh to seas.upenn.edu and use the turnin command to submit your files for grading:

```
% turnin -c cis530 -p hw4 hw4_code_yourpennkey.py nouns.png verbs.png 1612890.alt
536101.alt 88246.alt
```

This should print out a confirmation message. If you are prompted for a section name, type "ALL". You can run turnin multiple times before the deadline. Each time you run turnin, it overwrites your previous submission for that assignment. You can check that the homework was submitted successfully:

```
% turnin -c cis530 -v
```

This will show you the list of file(s) you have submitted.

Data

The data for this homework is accessible in `/home1/c/cis530/hw4/`. We will use Starbucks announcements for this homework, it is located at the `data/` subdirectory. The `small_set/` subdirectory contains three

samples from the announcements to be played with in question 2. You will also find other files such as Google's universal POS files and GraphViz samples, they will be explained later.

1 Visualizing Dependencies

Based on a subset of the Starbucks data, we will find out the top 20 nouns and verbs, and visualize the dependency relationship between these words. We will also do word similarity as a WordNet warmup.

1.1 Universal POS Tagset (2 points)

The part-of-speech tags from the CoreNLP tool (which you have used in homework 3) corresponds to the Penn Treebank tags. You can view the Penn Treebanks at `http://bulba.sdsu.edu/jeanette/thesis/PennTags.html`. As you may have noticed, there can be many types of nouns and verbs. Thanks to Google's effort of uniting the world's language, we can map every part of speech in Penn Treebank into one of the Google's universal tagset. You can find more information at `http://code.google.com/p/universal-pos-tags/`.

For the purpose of this homework we will use a slightly modified version of their mapping, located at `/home1/c/cis530/hw4/en-ptb-modified.map`. To view the full Universal Tagset, read the README file under `/home1/c/cis530/hw4/universal_pos_tags.1.02`.

(2 points) Write a function `get_tag_mapping(map_file)` that takes in this mapping file, and produce a dictionary, whose keys are Penn Treebank tags, and values are their corresponding universal tag.

1.2 Getting the top nouns and verbs (15 points)

Now we are ready to get the top 20 nouns and top 20 verbs from our corpus, and we will continue to use Stanford's CoreNLP to do part-of-speech tagging and lemmatization.

1. Do part-of-speech tagging and lemmatization on all files under `data/` and get their xml files from CoreNLP.

2. (5 points) Write a function `get_all_sent_tok(xml_files)` that takes in a list of xml files and returns a list of all sentences from these files; each sentence returned is a list of tuples of the form `(word, lemma, part-of-speech)`. Make sure to lower-case all words and lemmas upon returning.

   ```
   >>> get_all_sent_tok(xml_file_list)
   [[tuples of sentence 1],[tuples of sentence 2],...]
   ```

3. (4 points) Write a function `get_nounverb_lemma_dict(tok_sents, tagmap)`, where `tok_sents` is a list of sentences in the format of the above question, tagmap is what you get from `get_tag_mapping`, and returns a dictionary whose keys are all nouns and verbs (that are also in Word Net) occurred in the text in their raw form, and the values are the lemmas these words correspond to.

   ```
   >>> get_nounverb_lemma_dict(tok_sents, tagmap)
   {'shares':'share', 'coffee':'coffee', ...}
   ```

4. (6 points) Write a function `get_top_nouns_verbs(tok_sents, tagmap, n)` that takes a list of tokenized sentences from 1.2.2 and a tag map, return a tuple of the n most frequent nouns and verbs (in their lemmas) that are not functional words, and can also be found in WordNet. The list of functional words is located at `/home1/c/cis530/hw4/funcwords.txt`

   ```
   >>> get_top_nouns_verbs(xml_files, tagmap, n)
   >>> ([list_top_n_nouns],[list_top_n_verbs])
   ```

1.3 Path Similarity in WordNet (13 points)

WordNet is a built-in part of NLTK, so chapter 2 of the NLTK book (particularly the section on WordNet) will be very helpful. Also, be sure to check out the functions and variables in the following NLTK modules:

http://nltk.googlecode.com/svn/trunk/doc/api/nltk.corpus.reader.wordnet-module.html
http://nltk.googlecode.com/svn/trunk/doc/api/nltk.corpus.reader.wordnet.Synset-class.html
http://nltk.googlecode.com/svn/trunk/doc/howto/wordnet.html

In this problem we will find out how to compute the similarity between two words using path similarity, and do sense disambiguation using their gloss and context.

1. (5 Points) First we need to get the context from words. For each word, we define its context be all unique words in the sentences the word appears in. We are doing this because the word can appear in a sentence in many forms. Write a function `get_context(lemmas, tok_sents)` where `lemmas` is a list of unique word lemmas we are interested in getting their context, `tok_sents` is a list of tokenized sentences from 1.2.2, and return a dictionary. The keys of the dictionary should be the input words, and the values should be a set of unique non-functional words that are considered the key's context.

   ```
   >>> get_context(lemmas, tok_sents)
   >>> {'detail':set([store, grocery...]),
    'supermarket':set(['deal','distribute',...]),...}
   ```

2. (8 points) You have learned how to compute the similarity between two words using the shortest path between them through the hypernym tree. This similarity method is implemented in NLTK as the function `path_similarity`.

   Write the function `get_path_similarity(word1, context1, word2, context2, pos)` that computes the word similarity between two words given their part-of-speech. You can assume pos is either "verb" or "noun". Now each word can have multiple synsets to which it belongs, and we will disambiguate them by picking the synset whose gloss is most similar to the word's context using cosine similarity. If the similarity score is strictly zero, then use the first sense of the word.

   ```
   >>> get_path_similarity(word1, context1, word2, context2, pos)
   >>> 0.1647
   ```

1.4 The Stanford Parser

We will be using the Stanford Dependency Parser (`http://nlp.stanford.edu/software/lex-parser.shtml`) in order to extract syntactic dependencies from the articles. We have provided a specially modified set of Python bindings for a version of the Stanford Parser installed on Eniac.

NOTE: This homework uses a specially modified version of this library. Use only the version on Eniac.

The following code snippet shows you how to extract dependency links from a sentence:

```
>>> from stanford_parser.parser import Parser
>>> p = Parser()
Loading parser from serialized file /home1/c/cis530/Software/python2.6/site-pack
ages/stanford_parser/stanford-parser-2010-08-20/../englishPCFG.July-2010.ser ...
 done [0.7 sec].
>>> s = "Pick up the tire pallet near the truck."
>>> deps = p.parseToStanfordDependencies(s)
```

```
# Flatten the dependency tree and organize the dependencies in the format:
# (relation, governor, dependent).
>>> print [(r, gov.text, dep.text) for r, gov, dep in deps.dependencies]
[('prt', 'Pick', 'up'), ('det', 'pallet', 'the'), ('nn', 'pallet', 'tire'), ('do
bj', 'Pick', 'pallet'), ('det', 'truck', 'the'), ('prep_near', 'pallet', 'truck'
)]
```

1.5 Performing Parsing (14 points)

1. (4 Points) Write a function `dependency_parse_files(filelist)` that loads and parses all of the sentences (as strings) in each document in the files specified by `filelist`, and returns the entire set of dependencies over all of the documents in a single list of triplets in the form `(relation, governor, dependent)`. Be sure to lower-case every word in every triplet.

   ```
   >>> dependency_parse_files(['file1.txt','file2.txt'])
   >>> [('prt', 'pick', 'up'), ('det', 'pallet', 'the'),...]
   ```

2. (5 Points) Write a function `get_linked_words(word, word_list, pos, dependency_list, lemma_dict)` that returns a list of all unique lemmas w such that the w appears as either the governor or dependent of a relation in `dependency_list` that also involves `word`. Note here items in `dependency_list` are not lemmatized. If a word in dependency list cannot be found in the lemma dictionary, just skip the word (for we know it is not in WordNet by the method we pre-processed our data to get the lemma dictionary).

3. (5 Points) Write a function `get_top_n_linked_words(word, word_list, pos, dependency_list, lemma_dict, context_dict, n)` that returns the top n words in `word_list` (ranked by similarity score from the previous problem) linked to `word`, according to `dependency_list`. Sort the returned words in descending order of similarity.

   ```
   >>> get_top_n_linked_words('coffee', ['store','company',...], 'noun', dependency_list,
   lemma_dict, context_dict, 5)
   >>> ['store','seattle',...]
   ```

1.6 Using Graphviz (16 points)

We will use the Graphviz tool to visualize the relationships between words that you have computed from the previous sections. This tool, which is already installed on Eniac, allows you to create a graph given some relationships between entities. To do this, we must prepare an input file. The input file to Graphviz is a file where each line specifies two words that are related. A sample input file `/home1/c/cis530/hw4/sample.viz` and sample graph file `/home1/c/cis530/hw4/sample.png` are provided on Eniac.

To use graphviz to convert a graph file `sample.viz` to a `.png` file type the following at the command prompt:

```
dot -Tpng sample.viz -o sample.png
```

It would be easiest to convert the files on Eniac, but if you want to install Graphviz on you own machine, you can select the appropriate version to download at `http://www.graphviz.org`

1. (2 Points) Write a function `create_graphviz_file(edge_list, output_file)`. The parameter `edge_list` is a list of string tuples. The function should create a file with the name `output_file` which contains the word links in the Graphvis input format. For example:

   ```
   >>> create_graphviz_file([('dog', 'cat'), ('dog', 'computer'), ('cat', 'computer
   ')], 'test.gr')
   ```

Should create an output file called `test.gr`, containing the following:

```
graph G {
dog -- cat;
dog -- computer;
cat -- computer;
}
```

2. (14 Points) Visualize and produce two graphs, each for the top 20 nouns and verbs found in their lemmatized form, and name them `nouns.png` and `verbs.png`. The edges in the graph should be dependency relationship amongst all words in the same group (noun or verb). For a particular word, consider the top 5 words with which it has a dependency relationship (this order should be obtained from path similarity).

2 Alternative Text

2.1 Extended Lesk Similarity (20 points)

eLesk is a similarity measure over WordNet glosses. The original paper for the eLesk algorithm can be found at `http://www.d.umn.edu/~tpederse/Pubs/ijcai03.pdf`. A very detailed description of how to calculate the similarity can be found in the 3rd paragraph of section 3.2 of the paper. Here we will consider a maximum sequence of length 2. Use the list of functional words is located at `/home1/c/cis530/hw4/funcwords.txt`. For the relationships, we will use the gloss of the word itself, and the glosses of its hyponyms (Hint: you can treat a word's hyponym glosses as a single gloss by simply concatenate them together).

1. (10 points) Write a function `calc_gloss_sim(gloss1, gloss2)` that returns the eLesk similarity score between two glosses.

2. (10 points) Write a function `get_lesk_similarity(word1, context1, word2, context2, pos)` that returns the eLesk similarity between word1 and word2, both having the same part-of-speech pos. Just as we did in calculate path similarity, use the disambiguated sense with their contexts.

2.2 Substituting Nouns and Verbs (20 points)

1. (10 points) Here is how we will be finding alternatives for a word in WordNet. Given a word, get the disambiguated synset. For that particular synset, if it has a parent then get one of its siblings at random. If not, get one of its children at random. (In rare cases where there are multiple parents, get one at random; if the word has no parent or child, return the word itself). Write a function `get_random_alternative(word, context, pos)` that does exactly so, and return the alternative verb found. (Please make sure that you set the random seed to zero at the beginning of this function).

```
>>> get_random_alternative('reported', set(['company','revenues',...]), 'noun')
>>> 'presentation'
```

2. (5 points) Write a function `gen_alternative_words(wordlist, tok_sents, pos)`, that given a list of words and tok_sents from 1.2.2, return a list of tuples whose ith entry is of the tuple (A_i,B_i,C_i), where:

   A_i is the original word

   B_i is the alternative word

   C_i is the eLesk similarity score between the two.

Here we will use the entire document as context. Since our ultimate goal is to substitute nouns and verbs in a document with their alternatives, it is more convenient to use words in their raw form in this case, instead of the lemmas. Rank the result in decreasing order of similarity.

3. (5 points) Write a function `gen_alternative_text(textfile, xmlfile, tagmap)` that takes in a text file, a pos-tagged xml file, and a tagmap you have implemented from Google's universal tag set in question 1, and return the text such that every noun and every verb (that is in WordNet) in the text is substituted by a random alternative. Do this for every document inside the `small_set/` directory. Note that these documents are also inside `data/`, so you don't need to parse them again. Does the alternative text makes sense? Does a larger similarity score correspond to a better alternative? Rank your opinion on a scale of 1-5, 1 being that you think the alternative text are very problematic in terms of wording, and 5 being that the text could easily have been taken from a newspaper (all sentences makes sense and correct, but please ignore tense, plural etc). Write down as comments in your code.

   Submit the alternative text files and name them `.alt`, for example, 1612890.alt