CS366 Assignment 4 Due: 2020-04-06 11:59PM

1. Using Similarity Measures for WSD

Background

Please review the class slides and readings in the textbook on lexical semantics, including WordNet and word sense disambiguation. Also, please read the article in the assignment repository describing Resnik's word sense disambiguation approach in detail.

For additional information on NLTK's WordNet API and information content measures, see the NLTK WordNet HowTo. You may also want to consult the NLTK Word corpus reader information.

Computing Semantic Similarity

✎ Based on the examples in the text and the article, write a procedure `resnik_similarity` that implements Resnik's Wordnet-based similarity measure. The procedure should take two words and return the similarity of the most similar sense pair. Resnik's similarity measure relies on two components:

1. the Wordnet taxonomy, and
2. a corpus-based based information content measure.

Use the NLTK Python implementation of the Wordnet API for this assignment.

NOTE: You may use the API to access components of Wordnet, extract synsets, identify hypernyms, etc. You may not use the methods that directly implement the similarity measure or the identification of Least Common Subsumer. You must implement those functions yourself as procedures for the similarity calculation.

You may use accessors such as `common_hypernyms` and `information_content`. If you have questions about the admissibility of a procedure, please ask!

The NLTK installation provides a number of resources for information content calculation, including frequency tables indexed by Wordnet offset and part-of-speech. For consistency and quality, you should use `ic-brown-resnik-add1.dat`, which derives its counts from the Brown Corpus. The file includes fractional counts for ambiguous words (aka Resnik counting) with *add1 smoothing* to avoid zero counts for words not in Wordnet. You may use this source either through the NLTK API or directly through methods that you implement yourself (note that for the latter case, the file is included in the repository).

Performing Word Sense Disambiguation

Now you can implement a word sense disambiguation procedure that employs the Resnik similarity measure you created. The procedure will select the preferred sense based on the similarities between the senses of the probe word and the senses of the nouns in the context. **NOTE:** You do not need to select senses for all words, only for the probe word; this is a simplification of the word group disambiguation model in the paper.

Data

The following files are provided:

- `wsd_contexts.txt`: File of probe words with disambiguation word grouping lists. Each line is formatted as:

  `probe_word <TAB> word_grouping`, where
  - `probe_word` is the word to disambiguate
  - `word_grouping` is the comma-separated word list that serves as disambiguation context
- `wsd_contexts.txt.gold`: Corresponding file with gold standard sense tags, in which the sense id and gloss are prepended to the original line.

NOTE: The Resnik WSD approach may not be able to disambiguate these words correctly. You will probably achieve about 60% accuracy overall. The first several instances are much easier than the later ones.

✎ Word sense disambiguation program

Create a program named `wsd.py` with the following parameters:

- *information_content_file*: This should specify the name of the information content file. Here, you should be using `ic-brown-resnik-add1.dat`.
- *wsd_test_file*: The file containing ”probe-word, context-words” pairs on which to evaluate your system. Use the wsd_contexts.txt file specified above.
- *results.out*: The file in which to put the results.

Your program should:

- Load the information content measure.
- For each (word,context) pair:
  - Use your Resnik similarity function with the Wordnet API to compute the preferred Wordnet sense for the probe word given the context.
  - On a single line, print the similarity between the probe word and each context word in the format `W1,W2,similarity`
  - Print out the preferred sense, by synsetID, of the word.

NOTE: The procedure needs to work only on nouns.

✎ Write-up

Describe and discuss your work in a page or two. Include problems you came across and how (or if) you were able to solve them, any insights, special features, and what you learned. Give examples if possible. If you were not able to complete parts of the project, discuss what you tried and/or what did not work. This will allow you to receive maximum credit for partial work.

2. Using classifiers for WSD

This exercise explores the use of machine learning for word sense disambiguation. You will use NLTK to build Naive Bayes classifiers to disambiguate selected words, using data from Senseval-2. You will not be doing any coding for this exercise; all code is provided, and you will simply run it and evaluate results.

Data

The Senseval-2 corpus consists of text from a mixture of places, including the British National Corpus and the *Wall Street Journal* section of the Penn Treebank. Each word in the corpus is tagged with its part of speech, and the senses of the following target words are also annotated: the noun *interest*; the verb *serve*; and the adjective *hard*. The senses used to annotate each target word come from WordNet and are shown in the table below.[1]

Word	**Sense**	**Definition**	**Example**
interest	interest_1	a sense of concern with and curiosity about someone or something	"an interest in music"
	interest_2	a reason for wanting something done	"in the interest of safety"
	interest_3	the power of attracting or holding one's attention (because it is unusual or exciting etc.)	"they said nothing of great interest"
	interest_4	a fixed charge for borrowing money; usually a percentage of the amount borrowed	"how much interest do you pay on your mortgage?"
	interest_5	(law) a right or legal share of something; a financial involvement with something	"they have interests all over the world"
	interest_6	(usually plural) a social group whose members control some field of activity and who have common aims	"the iron interests stepped up production"
hard	hard_1	not easy; requiring great physical or mental effort to accomplish or comprehend or endure	"a hard task"
	hard_2	dispassionate	"a hard bargainer"
	hard_3	resisting weight or pressure	"a hard rock"
serve	serve_2	do duty or hold offices; serve in a specific function	"she served in Congress for two terms"
	serve_6	provide (usually but not necessarily) food	"we serve meals for the homeless"
	serve_{10}	work for or be a servant to	"may I serve you?"
	serve_{12}	be sufficient; be adequate, either in quality or quantity	"nothing else will serve"

You will use the code in `wsd_code.py` for this exercise. Look at the code to get an overview of what it does. NOTE: `help(...)` is your friend:

- `help([class name])` for classes and all their methods and instance variables
- `help([any object])` likewise
- `help([function])` or
- `help([class].[method])` for functions / methods

[1]These are subsets of the full set of senses defined in WordNet, which is why the numbering of senses is not (always) contiguous.

This code allows you to run, train and evaluate a range of Naive Bayes classifiers over the corpus to acquire a model of WSD for a given target word from Senseval-2. Note that the names of the internal files containing the information for each Senseval-2 word has the format `<word>.pos`, which is what will be used to reference each word from within the Python code.

Once you run `wsd_code.py` you have access to all its methods. The session below shows how to use the method `senses()` to list the possible senses of the word *hard* and how to retrieve the set of all sense-tagged instances of this word. After using the method `instances()`, the value of `hard` is a list containing all the instances of *hard* in the corpus, and we can retrieve the first instance as `hard[0]`. An instance has four different attributes:

1. `word` specifies the target word together with its syntactic category; for example, `hard-a` means that the word is *hard* and that its category is *adjective*;
2. `position` gives its position within the sentence (starting with position 0);
3. `context` represents the sentence as a list of pairs, each consisting of a token and its tag;
4. `senses` is a tuple, each item in the tuple being a sense for the target word; typically, this tuple consists of only one argument, but there are a few examples in the corpus where there is more than one, representing the fact that the annotator couldn't decide which sense to assign to the word; for simplicity, we are going to ignore any non-first arguments to the attribute `senses`.

```
>>> senses('hard.pos')
['HARD3', 'HARD2', 'HARD1']
>>> hard = senseval.instances('hard.pos')
>>> hard[0]
SensevalInstance(word='hard-a', position=20, context=\[('``', '"'), ('he', 'PRP'),
('may', 'MD'), ('lose', 'VB'), ('all', 'DT'), ('popular', 'JJ'), ('support', 'NN'),
(',', ','), ('but', 'CC'), ('someone', 'NN'), ('has', 'VBZ'), ('to', 'TO'),
('kill', 'VB'), ('him', 'PRP'), ('to', 'TO'), ('defeat', 'VB'), ('him', 'PRP'),
('and', 'CC'), ('that', 'DT'), ("'s", 'VBZ'), ('hard', 'JJ'), ('to', 'TO'),
('do', 'VB'), ('.', '.'), ("''", "''")\], senses=('HARD1',))
```

You can see the 100 most frequent words that occur in the context of a given sense of a word (here, sense 1 of *hard*) as follows:

```
>>> instances1 = sense_instances(senseval.instances('hard.pos'), 'HARD1')
>>> features1 = extract_vocab_frequency(instances1, n=100)
>>> print (features1)
[('harder', 475), ('time', 316), ('get', 224), ('would', 224), ('find', 206), ('
    make', 195), ('very', 187), ('out', 175), ('people', 165), ('believe', 154), ('
    even', 152), ('hardest', 139), ('going', 136), ('much', 130), ('like', 127), ('
    just', 121), ('now', 111), ('way', 107), ('imagine', 105), ('see', 104), ('may'
    , 103), ('other', 103), ('know', 103), ('could', 96), ('into', 93), ('many',
    92), ('new', 91), ('tell', 88), ('these', 85), ('come', 85), ('too', 83), ('
    without', 82), ('good', 81), ('part', 79), ('take', 78), ('think', 77), ('work'
    , 76), ('thing', 75), ('really', 74), ('go', 73), ('two', 73), ('those', 71), (
    'makes', 70), ('back', 70), ('things', 70), ('still', 69), ('made', 67), ('last
    ', 67), ('having', 66), ('first', 66), ('most', 64), ('our', 64), ('here', 63),
     ('keep', 62), ('life', 62), ('enough', 62), ('making', 61), ('lot', 60), ('
    long', 58), ('year', 58), ('over', 58), ('getting', 57), ('only', 55), ('off',
    55), ('san', 55), ('home', 52), ('understand', 52), ('day', 52), ('why', 51), (
    'through', 51), ('might', 51), ('sometimes', 50), ('times', 50), ('something',
    49), ('little', 49), ('since', 48), ('does', 48), ('game', 48), ('kind', 47), (
    'while', 46), ('put', 46), ('down', 45), ('before', 45), ('children', 45), ('
    job', 45), ('want', 43), ('become', 43), ('often', 43), ('especially', 43), ('
    right', 42), ('place', 42), ('around', 41), ('found', 41), ('once', 41), ('
    anything', 40), ('doing', 40), ('women', 40), ('ever', 40), ('finding', 39), ('
    same', 39), ('someone', 39)]
```

Make sure you understand the representation of instances. Look at a few more instances of *hard* and then do the same for *interest* and *serve* (by replacing `'hard.pos'` with `'interest.pos'` and `'serve.pos'`, respectively).

Baseline classifiers

Before building sense classifiers for *interest, hard* and *serve*, it is necessary to establish baselines for comparison. Two commonly used baselines are the *random* baseline and the *majority* baseline. The random baseline is the accuracy that we can expect from guessing senses randomly–i.e., if there are 5 senses, the random baseline is 1/5 or .2; if there are 3 senses the random baseline is 1/3 or .33, etc.

The majority baseline is the accuracy obtained if the most frequent sense is always assumed to be the answer. You can derive the majority baseline for *hard* using NLTK methods as follows:

```
>>> import nltk
>>> hard_dist = nltk.FreqDist([i.senses[0] for i in senseval.instances('hard.pos')
    ])
>>> hard_dist
FreqDist({'HARD1': 3455, 'HARD2': 502, 'HARD3': 376})
>>> hard_baseline = hard_dist.freq('HARD1')
>>> hard_baseline
0.797369028386799
```

The method `nltk.FreqDist()` is used to compute the distribution of different senses of *hard* in the Senseval corpus. The output is stored in the variable `hard_dist` and shows that 'HARD1' has 3455 occurrences, while 'HARD2' and 'HARD3' occur only 502 and 376 times, respectively. Hence, by always guessing that *hard* has the first sense, we reach an accuracy (on this sample) equal to 3455 / (3455 + 502 + 376) = 0.797, or 79.7%. You should modify the code above to compute the majority baselines for *interest* and *serve*.

Naive Bayes classifiers

Compare the performance of different classifiers that perform word sense disambiguation, using the method `wsd_classifier()`, which must have at least the following arguments specified:

1. a trainer, in this case `NaiveBayesClassifier.train`
2. a target word: 'hard.pos', 'interest.pos' or 'serve.pos'
3. one of two feature representations, generated by these methods in `wsd_code.py`:
   - `wsd_word_features`: a feature set based on the set S of the n most frequent words that occur in the same sentence as the target word w across the training corpus (you can specify the value of n; default is 300). For each occurrence of w, `wsd_word_features` represents its context as the subset of those words from S

that occur in w's sentence. By default, the words that are specified in the variable `STOPWORDS` are excluded from the set S of most frequent words.You can add to or re-define this list by modifying the program. To see the words that are excluded by default, use `STOPWORDS`.

- `wsd_context_features`: a collocational feature representation that represents the context of a word w as the sequence of m pairs *(word,tag)* that occur before w and the sequence of m pairs *(word,tag)* that occur after w. You can specify the value of m (e.g., m=1 means the context consists of just the immediately prior and immediately subsequent word-tag pairs); otherwise, m defaults to 3.

The method splits the data available for a target word into a *training set* and a *test set*, trains a Naive Bayes classifier on the training set, and evaluates its accuracy on the test set. Here is how to use the method to evaluate the bag-of-words model for the target word *hard* :

```
>>> import nltk
>>> nb_hard = wsd_classifier(nltk.NaiveBayesClassifier.train, 'hard.pos',
    wsd_word_features)
Reading data...
Senses: HARD1 HARD2 HARD3
Training classifier...
Testing classifier...
Accuracy: 0.8108
```

✎ Use `wsd_classifier` to train a classifier that disambiguates *hard* using `wsd_context_features`. Build classifiers for *line* and *serve* as well, using the word features and then the context features. Provide a short write-up addressing the following:

1. Evaluate both feature representations for all three target words. Which is more accurate for disambiguating `hard.pos`, `wsd_context_features` or `wst_word_features`? Does the same hold true for `line.pos` and `serve.pos`? Why do you think that might be? Why might it not be fair to compare the accuracy of the classifiers across different target words?
2. Compare the results for the different classifiers, including your random and majority baselines.

Rich features vs. sparse data

In addition to being able to choose between `wsd_context_features` vs. `wsd_word_features`, you can also vary the following:

- In `wsd_context_features`, you can vary the number of word-tag pairs before and after the target word that you include in the feature vector. This is done by specifying the argument distance to the function `wsd_classifier`. For example, the following creates a classifier that uses two words to the left and right of the target word:

```
>>> wsd_classifier(NaiveBayesClassifier.train, 'hard.pos',
    wsd_context_features, distance=2)
```

- In `wsd_word_features`, you can vary the words that are excluded from the set of stopwords and the size of the set of stopwords. For example, the following results in a model which uses the 100 most frequent words, including stopwords:

```
>>> wsd_classifier(NaiveBayesClassifier.train, 'hard.pos', wsd_word_features,
    stopwords_list=[], number=100)
```

✎Build several WSD models for `'hard.pos'`, including at least the following:

- for the `wsd_word_features` version, vary the number between 100, 200 and 300, and vary `stopwords_list` between [ ] (i.e., the null list) and STOPWORDS
- for the `wsd_context_features` version, vary the distance between 1, 2 and 3, and vary `stopwords_list` between [ ] and STOPWORDS

Provide a writeup addressing the following questions:

- How does setting number to less than 300 affect the word model? How does reducing the context window before and after the target word to a number smaller than 3 affect the model?
- How does including stopwords in the word model affect overall performance? **Hint:** For each sense of *hard*, construct a list of all its instances in the training data using the function `sense_instances`. Then, call, for example, `extract_vocab_frequency(instances1, stopwords=[], n=100)` and compare this with what you get for instances2 and instances3.
- It seems slightly odd that the word features for 'hard.pos' include *harder* and *hardest*. Try using a stopwords list including these words. Is the effect what you expected? Can you explain it?

Error analysis

After analyzing the performance of a classifier, it is useful to perform an error analysis. By providing additional arguments to `wsd_classifier()`, you can generate a confusion matrix as well as a printout of all the errors.

```
>>> nb_hard = wsd_classifier(nltk.NaiveBayesClassifier.train,'hard.pos',
    wsd_word_features,distance=3,confusion_matrix=True,log=True)
Reading data...
Senses: HARD1 HARD2 HARD3
Training classifier...
Testing classifier...
Accuracy: 0.8016
Writing errors to errors.txt
| H H H |
| A A A |
| R R R |
| D D D |
| 1 2 3 |
------+---------------+
HARD1 | <692>   7   3 |
HARD2 |   84  <2>   2 |
HARD3 |   75    1 <1> |
------+---------------+
(row = reference; col = test)
```

The relevant arguments are `confusion_matrix=True` and `log=True`, but note that you must also insert the argument `distance=3` to get the right interpretation. The errors are printed to an external file called `errors.txt`[2]

✎ Do an error analysis of the best performing classifier for each target word, answering the following questions:

1. Using the confusion matrix, identify which sense is the hardest one for the model to estimate.
2. Look in `errors.txt` for examples where that hardest word sense is the correct label. Do you see any patterns or systematic errors? If so, can you think of a way to adapt the feature representation to improve the model?

[2]The file is put in your working directory; to determine the working directory, use the command `import os` followed by `os.getcwd()`.

Submission

This assignment will be submitted by including it in your GitHub classroom repository. You should include the following:

1. Your code for *resnik_similarity*
2. Your code for *wsd.py*
3. A plain text file or PDF document with your writeup for Question 1
4. A plain text file or PDF document with your answers to all three parts of Question 2
5. A README file describing the files and their contexts