

Natural Language Understanding CS 224U / LING 188 / LING 288

Word-sense Disambiguation Programming Homework

Due: Wednesday October 18, 2007, 5 pm

Read this assignment soon, and especially if you are less familiar with programming, start working on it, discover stumbling blocks, and ask questions early. If anything is unclear or you need some kind of assistance, feel free to send a question to cs224u-aut0708-staff@lists.stanford.edu.

The aim of this assignment is to write a system that does context-based word sense disambiguation. The materials that you need for this assignment are in the following directory in AFS:

```
/afs/ir/class/cs224u/wsdp
```

If you need more information on Unix computers at Stanford, look here:

```
http://www.stanford.edu/services/unixcomputing/
```

Assignment Description

You will write a program that will disambiguate the senses of an ambiguous word. You will be provided with tagged training data and test data for several words. You should probably begin by making a straightforward implementation of some previously-tried technique for word sense disambiguation (some suggestions follow), and then seeing if you can improve it - the aim being to make this program as good at word sense disambiguation as possible. Your final submission should consist of a single system that performs as well as possible, or applies an interesting machine learning technique to disambiguation. The program may require other lexical resources that you make use of (thesaurus, WordNet, etc.). The only hand-tagged training instances that it should use are the ones supplied.

The Data

The data for developing your program is located in `/afs/ir/class/cs224u/wsdp/data`.

For each word in `data/`, there will be a training file of examples with one sentence/context per line. In each line of the tagged training data (all filenames ending in `.train`), exactly one word to be disambiguated will be marked with an underscore, and followed by a sense identifier. The identifier will simply be an integer denoting the correct sense number for that occurrence. There will be no other underscores in the data. (You may assume that there are no more than 30 senses for any given word; however, we will look more favorably on code that does not depend upon a hard-wired maximum number of senses.) There may be a file `.senses` that gives definitions for the different senses in human-readable form.

For the test data (all filenames ending in `.test`), the senses will be marked with only an underscore. The idea is that you should use each word's tagged training data to learn about the word's senses, and then predict the senses for the underscored words in the test data.

For example:

```
Training data format: bank_2  
Test data format: bank_
```

We have chosen the word data to represent the range in the amount of training data we will provide for each word during evaluation. You will notice that some words have more training data than others.

Your program should take two command-line arguments, a training filename and a test filename, and output for the testing data which sense (integer) it regards the example as (one per line). **Your program should not output any additional information.** The training and testing data will guarantee a context of at least 5 "orthographic words" to the left and right of the test word, but note that punctuation marks, function words, etc. can all count as orthographic words, and that in some cases we have padded the document with the "dummy" words "xtrainx" in order to achieve this 5 word context. You may choose to ignore these dummy words. The test word will be present at least once in each line. Even if the word appears more than once in the line, you should still disambiguate the occurrence marked with an underscore. On a single run of the word sense disambiguation program, it will be disambiguating only one word.

Note that, in context, the actual word to be disambiguated may be in a morphologically inflected or capitalized form of the word in the training data. The correct answers for each line of the test data are in the .ans files - of course, these are to be used only for evaluating the program's performance! You may wish to use some words or part of the data for each word as a basis for exploratory data analysis and feature engineering, and to reserve other words or some of the data for all words as development test data. Or you could combine the training and test data to do cross-validation. It's up to you, though you should avoid overtraining your system to these test sets. Many of the data sets have senses for only one part of speech, but not all.

We will test the program on different words, with test and train files in the same format as those provided for development.

There are a few bits of code available in the `src` directory. The file `baseline.pl` is a complete WSD system in Perl, which implements the most frequent sense baseline method. The file `eval.pl` can evaluate your WSD system, including printing a handy confusion matrix. This command evaluates the baseline system on the interest data (when run from the `cs224u/wsdp` directory):

```
src/baseline.pl data/interest.train data/interest.test | src/eval.pl data/interest.ans
```

You should ensure that your program works correctly when piped through the `eval.pl` program like this!! That's how we will evaluate systems.

A couple of Java classes are also available to help you get started. These are in `/afs/ir/class/cs224u/wsdp/src`. The following command evaluates a worse-than-baseline WSD system:

```
java -cp src WSD data/interest.train data/interest.test | src/eval.pl data/interest.ans
```

The file `WSD.java` contains the main function that reads the training and test data, and may provide a convenient starting framework. Actually, it's pretty bad code. I wouldn't use it, but you're welcome to. You're not constrained to use Perl or Java. You can use any language that we can run on the Stanford Unix Computing machines. Go on, write it in OCaml. (However, you're only likely to be able to get programming help if you choose a language that the TA knows....)

Note: Some of the data provided is from the Senseval competitions. We will be using further data from these competitions to test your programs. It is our intention that this data is previously unseen; therefore, *do not seek out and use extra Senseval data when creating your program*. You should use only the WSD data provided (but feel free to use other lexical or corpus resources, such as newswire, WordNet, etc.)

Submission Instructions

For the final submission, you need to submit both your program (electronically) and a written report.

Submission of the program code will be via a script. To submit your program, put all the needed files (including all source files, scripts and a Makefile, if necessary) in one directory and, from that directory, type:

```
/afs/ir/class/cs224u/bin/submit-wsdp
```

If your system is difficult to invoke, it'd be really sweet if you could include a Makefile and a Unix shell scripts named *runwsd* which takes two file arguments, as in the examples above.

The Report

The report should detail the method used by your program, the architecture of your program, testing you did, dead-end paths, and model revisions you pursued. What we're looking for in the grading is:

- A clear discussion of the algorithms/method used.
- A discussion of the testing you did and results you obtained. If you redivided the provided data into training and testing portions, mention it in the report.
- Data analysis, especially of errors, and insights you gained from that which you used to improve your program.
- Experiments and observations on the strengths and weaknesses of the machine learning technique(s) that you implemented.
- A discussion of alternatives or things you tried to improve performance, and how they fared.
- The performance of the system: it should perform decently above baseline performance (that simply assigns the most frequent sense in the training data to each word).
- Clean, intelligible code.
- Concise, clear reporting is highly preferred. Lengthy, rambling explanations is highly undesirable.

And in particular a clear (but brief) statement in answer to the following questions:

Question 1. The WSD program works better at disambiguating some words than others. What are the factors that seem to be determining performance, and what is their relative weight?

Question 2. Discuss the machine learning technique(s) that you implemented. If you implemented more than one, did they perform well on the same test sets? Did they depend on the amount of training data available? What were the major differences between the methods?

The report shouldn't be longer than 4 pages.

Contest!

We will run your systems on other test data. We will award a prize and bonus points of up to 10% to the best performing system or systems! (For this contest, we're just going to look at how systems perform on our data - we're not going to test whether superiority of one system over another is statistically significant.)

Please note that in general there isn't a linear relationship between performance and your grade. In general, we expect reasonable above baseline performance, but your grade is more dependent on the quality of your thinking and write-up than on the exact performance figure. For example, we would look favorably on someone who proposes a sensible-sounding different approach to word sense disambiguation, tries it out

and who finds out that it doesn't work that well in practice.

Basic Methods

Some machine learning techniques may work well with larger amounts of training data, but poorly with the kind of data we have. Experiment with these considerations before choosing your algorithms. Here are some of the kinds of methods that you could start off with:

1. A straight implementation of a Naive Bayes bag-of-words model of the kind used by Gale, Church, and Yarowsky (see Jurafsky and Martin Chapter 20, or one of several other sources, such as Manning & Schuetze Section 7.2.1.)
2. An n -gram language model. (Two things that suggest that this method does not perform too horribly are (i) Kaplan (1950) found that humans were almost as good at disambiguation having seen only one or two words on each side as the whole sentence, and (ii) about half of the decision list items in Yarowsky's (1995) approach appear to involve adjacent words (based on his examples).)
3. Using a naive Bayes model based on a thesaurus (Sec 7.3.2). It's not the one that Yarowsky/AT&T used, but a slightly extended version of the 1911 (!) printing of Roget's thesaurus (which is in the public domain) is contained in:

```
/afs/ir/class/cs224u/wsdp/resources/roget13a.txt
```

However, it would need a certain amount of parsing up....

4. Using a vector space model of contexts - a method widely used in Information Retrieval for measuring contextual similarity. Vector space measures are discussed towards the end of Ch 20 of J&M.
6. Using a memory-based learning / k -nearest neighbor approach.
7. Other ideas in Chapter 20 of Jurafsky and Martin or Chapter 7 of Manning and Schuetze, or papers cited there, such as the review of WSD systems in *Computational Linguistics* (Ide and Veronis 1998).
8. Any other supervised machine learning method with which you are familiar.

Things to think about:

- Using prior probability of senses
- Combining models
- Using word classes
- If you multiply too many probabilities, the numbers get too small and become zero on a computer.

You should probably use logs and add them.

- Probabilities frequently need smoothing. Zero probabilities are very harmful, since they say something is impossible. Even if you didn't happen to see a word in the context of a certain sense, you probably wouldn't want to say that that occurrence is impossible. The simplest method of smoothing is the "add one" method. If you're not familiar with it, you'll also want to look at Chapter 4 of Jurafsky and Martin at: <http://www.cs.colorado.edu/%7Emartin/SLP/Updates/4.pdf>. In it, you add one to each count, and then add the size of the vocabulary to the denominator (so that things are normalized and probabilities that sum to 1 still result). However, you may want to try using other smoothing methods to see if they improve your performance.

If there are other natural language process lexical resources or tools that you'd like to try incorporating, feel free to ask if they're available. There are a bunch of morphological analyzers, language model tools, and textual data that we could also make available.