Natural Language Processing CS224N/Ling237
Christopher Manning
Word-sense disambiguation project
Checkpoint: Wednesday April 24 2002, 5 pm
Due: Wednesday May 1 2002, 5 pm

Word-sense disambiguation and context-sensitive spelling correction

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 `cs224n-staff@lists`. However, before you do that, you're encouraged to look for any FAQs on the website:

`http://cs224n.stanford.edu`

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

`/afs/ir/class/cs224n/wsdp`

Assignment Description

Part A. The first part of this is a program that will disambiguate the senses of an ambiguous word. The program may require tagged training instances or other lexical resources that you make use of. 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. You are required to implement at least two different machine learning techniques. However, your final submission should consist of a single system that performs as well as possible.

Part B. The second part is to investigate whether these same techniques can be extended to context-sensitive spelling correction. The idea here is that one wants to correct spelling errors of the sort of {there, they're, their} or {weather, whether}, where all the choices are *some* correctly spelled word, but the right one has to be chosen. One could hope to do this by using the sentence context to decide.

Submission Instructions

There are two due dates for this assignment:

1) a checkpoint, one week before the assignment is due
2) the final due date

The intention of the checkpoint is to give you an indication of how well your program works on unseen data. We will test your code and give you your results; we will also post statistics about the class's performance as a whole, so that you can see how your results compare.
For the checkpoint, you are required to submit a working program electronically. No written report is necessary; however, please include a README file if there are any notes about your program that you wish to convey. See below for instructions on submitting your program electronically.

For the final project, you need to submit both your program (electronically) and a written report. Details for the report are found at the end of this handout.

Submission of the program code will be via a script. To submit your program, put all the needed files (including all source files and a Makefile, if necessary) in one directory on a Sweet Hall machine (or,

possibly, another machine with AFS on it and with you being correctly authenticated) and, from that directory, type:

```
/afs/ir/class/cs224n/bin/submit-wsdp-check (for the checkpoint)
/afs/ir/class/cs224n/bin/submit-wsdp       (for the final project)
```

If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-wsdp-check -replace (for the checkpoint)
/afs/ir/class/cs224n/bin/submit-wsdp -replace       (for the final
project)
```

> **You should include two Unix shell scripts named *runwsd* and *runspell* that compile and run your programs when executed. Each script should take two arguments: the first argument is the name of the training file, and the second argument is the name of the test file.**

runwsd should run the word sense disambiguation program and *runspell* should run the spelling correction program. If you use the same program for both, you should still have two shell scripts, but their contents will be identical. There are a couple of sample *runwsd* shell scripts in the `wsdp` directory – one called *runwsd* and one called *runwsd_example_java*. The first can be used to run the example given at the end of this handout, and the second simply shows a possible format for a shell script to compile and run a java program. On a single run of the word sense disambiguation program, it will be disambiguating only one word.

We will be running programs on further unseen data on the Sweet Hall systems, and your program should run without problems there.

The program can be written in a language of your choice, but Java is recommended, as we will only answer questions about Java code. In addition, there are some helper files available only in Java.

The Data

The data for developing your program is located in `/afs/ir/class/cs224n/wsdp/data.` The files in this folder consist of training data only. For each file, the first line will have on it the ambiguous word. Thereafter, sentences/contexts will be one per line.

For the tagged training data, the senses will be marked by an underscore and then a sense identifier. In the word sense disambiguation data, these identifiers will be integers. 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 will be no other underscores in the data. For the test data, the senses will be marked with only an underscore in real test data (though you may want to leave the senses marked in your validation data, so you can easily assess performance). Note that, in context, the actual word to be disambiguated may be a morphologically inflected or capitalized form of the word on the first line of the file.

> Training data format: bank_2
> Test data format: bank_

There are a few Java classes available to help you get started that you can use if you wish:

`/afs/ir/class/cs224n/wsdp/TestCaseChooser.java` can be used to convert a proportion of training data to test data format.

`/afs/ir/class/cs224n/wsdp/InstanceSet.java` and `/afs/ir/class/cs224n/wsdp/Instance.java` can be used to read in and tokenize a training file, and locate the ambiguous word in each instance.

The program should output for the testing data which sense (number) 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. If the word appears more than once in the line, you should disambiguate the occurrence marked with an underscore.

> Note: 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, *it will be considered an honor code violation to 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.)

The context-sensitive spelling program may be the same program or a variant one, as you choose. Basically, it is the same except:

- The first line of the training file will be a sequence of words, listing the confusion set to be corrected, separated by the "|" character (e.g. effect|affect).
- The "confused" word will appear as the above disjunct followed, in the training data, by an underscore and the correct spelling. (e.g. effect|affect_effect in training, but just effect|affect_ in testing data.) The output should be a word from that set, which is the "correction." Note that there is no indication of what the original was, so there's no issue of determining what the correct word is *given* the user's choice. You don't know the user's choice, or equivalently you must make the assumption that the user just picks one of the words uniformly at random. A real system must, of course, take into account what the user actually wrote!

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. To do this, you need to divide the provided data into training and testing portions.
- data analysis, especially of errors, and insights you gained from that which you used to improve your program.
- experiments and observations of the strengths and weaknesses of the two machine learning techniques 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.
- clean, intelligible code.

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 two (or more) machine learning techniques that you implemented. Did they perform well on the same test sets? What were the major differences between the methods?

Question 3. How did your program perform at the checkpoint? What improvements did you make after that performance?

Question 4. For the spelling error examples as well, you probably get different performance levels. For different kinds of spelling confusions, different kinds of information would seem most useful for disambiguating them. What kind(s) of information does your program use? In what circumstances is it helpful? In what sorts of cases would different sources of information be more helpful?

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.) But 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, and tries it out and who finds out that it doesn't work that well in practice.

The report shouldn't be longer than 5 pages.

As an example system, an (honest) baseline contestant (written in Perl) is available in

```
/afs/ir/class/cs224n/wsdp/src/baseline.pl
```

It always returns the most frequent sense in the training data (or sense 1 if no tagged training data is read in). It takes two arguments, the training and testing files. Line 1 of the training data has the word to train and test on. You can run it on some of the data used in last year's assignment using the commands below. (You can also feel free to use this extra data when developing your classifier – however, please note that last year's format may not be exactly the same as this year's format. In particular, the word to be disambiguated in the test cases is not marked with an underscore.)

```
cd /afs/ir/class/cs224n/wsdp/
./runwsd wordsense/drug.train wordsense/drug.test
```

To evaluate how well it does, try:

```
./runwsd wordsense/drug.train wordsense/drug.test | src/eval.pl
wordsense/drug.ans
```

The script `src/eval.pl` compares the output of a system (a list of senses, one on each line) with the real answers (stored in the .ans file) and provides a confusion matrix for your results. You can use this script to evaluate your own system, if you would like.

Basic Methods

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-based method of the kind used by Gale, Church, and Yarowsky (Section 7.2.1 of the text.)

2. An *n*-gram word 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/cs224n/pp2/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. Such vector space measures are discussed in both Section 8.5 and Chapter 15 of M&S.

5. A Bernoulli Naive Bayes model based on word presence/absence in the context rather than the model in Section 7.2.1 which uses a multinomial distribution over each word that *does* appear in the context. If you use this method, you probably need to do feature selection to achieve decent results. Mutual information between context words and senses can be used to identify good features.

6. Using a memory-based learning / k-nearest neighbor approach, as discussed in class.

7. Other ideas in Chapter 7 of M&S, 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 (as in equation (7.5) for Naive Bayes).
- 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, which you can find in Section 6.2.2. 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.