CS 224N / Ling 237

Assignment 1: Language Modeling

Due 11 April 2005

This assignment may be done individually or in groups of two. See the homework collaboration policy on the website (http://cs224n.stanford.edu/policies.html).

Please read this assignment soon and go through the Setup section to ensure that you are able to access the relevant files and compile the code. Especially if your programming experience is limited, start working early so that you will have ample time to discover stumbling blocks and ask questions.

1 Setup

On the Sweet Hall machines (such as elaine.stanford.edu), make sure you can access the following directories under /afs/ir/class/cs224n:

java/ : the Java code provided for this course
corpora/assignment1/ : the data sets used in this assignment

Copy the java/ directory to your local directory and make sure you can compile the code without errors. The code compiles under JDK 1.4, which is the version installed on the Sweet Hall machines.

To ease compilation, we have installed ant in the class bin/ directory. ant is similar in function to the Unix make command, but ant is smarter, is tailored to Java, and uses XML configuration files. When you invoke ant, it looks in the current directory for a file called build.xml which contains project-specific compilation instructions. The java/ directory contains a build.xml file suitable for this assignment. Thus, to copy the source files and compile them with ant, you can use the following sequence of commands:

```
cd ~
mkdir cs224n
cd cs224n
cp -r /afs/ir/class/cs224n/java java
cd java
/afs/ir/class/cs224n/bin/ant
```

(You can spare yourself the trouble of typing the full pathname for ant by adding it to your path, or by taking advantage of the symlink to ant located in the java/ directory.)

If you don't want to use ant, you are welcome to write a Makefile, or for a simple project like this one, you can just do

```
cd cs224n/java/
mkdir classes/
javac -d classes src/*/*/*.java
```

Once you've compiled the code successfully, you should test that you can run it. In order to execute the compiled code, Java needs to know where to find your compiled class files. As should be familiar to every Java programmer, this is normally achieved by setting the CLASSPATH environment variable. If you have compiled with ant, your class files are in java/classes, and the following commands will do the trick:

```
cd cs224n/java
setenv CLASSPATH ${CLASSPATH}:./classes
```

Now you're ready to run the test:

```
java cs224n.assignments.LanguageModelTester /afs/ir/class/cs224n/corpora/assignment1
```

If everything's working, you'll get some output about the performance of a language model being tested. The code is reading in the first 80% of the Penn Treebank (parsed WSJ text), discarding the trees, and feeding just the sentences to a language model implementation that we've provided (EmpiricalUnigramLanguageModel). This is a phenomenally bad language model, as you can see from the strings it generates. However, it shows the interface that the language models you'll write should implement (cs224n.langmodel.LanguageModel). The language model is trained on construction, by being passed a list of sentences. Note that these sentence lists are disk-backed, so doing anything other than iterating over them will be very slow, and shouldn't be attempted. A language model must respond to two important methods: getSentenceProbability, which scores a sentence, and generateSentence, which generates one. In the example language model, this is broken down into word-scoring and word-generating steps. This is usual, but you can use this structure or not as you like.

In cs224n.util there are a few utility classes that might be of use – particularly the Counter class. It makes dealing with word to count maps much easier.

2 Language Modeling

Take a look at the main method of LanguageModelTester.java, and its output. It loads several objects. First, it loads a training corpus of WSJ sentences from the Penn Treebank, as well as a validation (held-out) set of sentences and a test set from the same source (split 80% / 10% / 10%). Second, it loads a set of speech recognition problems (HUB). For each problem in HUB a SpeechNBestList object is created. These objects contain a correct answer and a set of candidate answers, along with acoustic scores for those candidates. This is typical of how rescoring works in speech recognizers: you have candidate recognitions, with some score of how much the acoustic model believes in the candidate based on the acoustic signal, and this estimate will be combined with a language model score.

Once these objects are loaded, an astonishingly bad language model is built from the training sentences (the validation sentences are ignored entirely). Then, several tests are run. First, the tester calculates the perplexity of the test WSJ sentences. Good numbers would be under 200; these aren't good numbers. Then, the perplexity of the HUB correct answers is calculated. This number will in general be much worse, since these sentences are drawn from a slightly different source, so our WSJ language models will be worse at predicting them. Note that language models can treat all entirely unseen words as if they were a single UNKNOWN token. This means that, for example, a good unigram model will actually assign a larger probability to each unknown word than to a known but rare word.

The third number produced is a word error rate (WER). The code takes the speech recognition problems' candidate answers, scores each candidate with the language model, and combines that score with a pre-computed acoustic score. The best-scoring candidates are compared against the correct answers, and WER is computed. The testing code also provides information on the

range of WER scores which are possible: the correct answer is not always on the candidate list, and the candidates are only so bad to begin with (the lists are pre-pruned n-best lists).

The final output is the result of generating sentences at random. Note that the provided language model's outputs aren't even vaguely like well-formed English.

Your job is to implement several language models of your choice that do a much better job at modeling English. You should be able to see the improvement both in terms of the perplexity and word error rate scores, and in terms of the generated sentences looking much better.

Here's what you should minimally build:

1. A well-smoothed unigram model. One good choice is to use a form of Good-Turing estimation, but you could also use absolute discounting.

2. Higher order bigram and trigram models. These must provide some means of interpolating between higher and lower order models. You could use either linear interpolation or Katz' backing off.

3. Some smoothing method which makes some use of the held-out data set. For instance, it could set weighting parameters in a linear interpolation.

Some other things you might try:

4. Implement some other ideas for smoothing speech language models. Possibilities might include Witten-Bell or Kneser-Ney smoothing, or Bayesian smoothed n-gram models. Good sources for copious detail about language modeling options are the papers by Goodman (2001) and Chen and Goodman (1998), which can be found on the class webpage.

5. Train on a larger dataset. The main method of LanguageModelTester accepts an optional second argument, which is the prefix of the filenames of the data files used. The default value is "treebank-sentences-spoken". At present, this is the only dataset available; however, we expect to make a much larger dataset available over the coming days.

3 Submitting the assignment

3.1 The program: electronic submission

Submission of the program code will be via a script. To submit your program, put all the needed files in one directory on a Sweet Hall machine (or, possibly, another machine with AFS on it and with you being correctly authenticated) and type:

```
/afs/ir/class/cs224n/bin/submit-hw1
```

If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-hw1 -replace
```

Make sure that you include all the source code for your programs. We will run programs on the Sweet Hall systems, so they should run without problems there. Please don't include large data files in your submission. Your code doesn't have to be beautiful but we should be able to scan it and figure out what you did without too much pain.

3.2 The report: turn in a hard copy

You should turn in a write-up of the work you've done, as well as the code. The write-up should specify what you built and what choices you made, and should include the perplexities, accuracies, etc., of your systems. **Your write-up must be submitted as a hard copy.** There is no set length for write-ups, but a ballpark length might be 4 pages, including your evaluation results, a graph or two, and some interesting examples. It would be useful to show a learning curve indicating how your system performs when trained on different amounts of data.

An important expectation is that for each language model you build (and in particular, the three minimally required models), you should show that the language model defines a proper probability distribution. That is, you should show (with equations and argument, as appropriate) that the model satisfies $\sum_w P(w|h) = 1$, where w is a word and h (for 'history') represents the words appearing before w.

The report should also include some error analysis – enough to convince us that you looked at the specific behavior of your systems and thought about what it's doing wrong and how you'd fix it.

4 Evaluation criteria

While you are building your language models, hopefully lower perplexity will translate into better WER, but don't be surprised if it doesn't. A best language-modeler title and a lowest WER title are up for grabs, but the actual performance of your systems is not the major determinant of your grade on this assignment (though good performance may suggest that you are doing something right).

What will impact your grade is:

- A discussion of the motivations for choices that you made and a description of the testing that you did.
- A clear presentation of the language modeling methods you used.
- Showing that your language models are proper probability distributions.
- The degree to which you can make sense of what's going on in your experiments through error analysis. When you do see improvements in WER, where are they coming from? Try to localize the improvements if possible. That is, figure out what changes in local modeling scores lead to WER improvements. (This will almost certainly require altering the testing code, and is strongly encouraged.)
- Your data analysis on the speech errors that are occurring. Are there cases where the language model isn't selecting a candidate which is clearly superior? What would you have to do to your language model to fix these cases? For these kinds of questions, it's actually more important to sift through the data and find some good ideas than to implement those ideas. The bottom line is that your write-up should include concrete examples of errors or error-fixes, along with commentary.

4.1 Extra credit

We will give up to 10% extra credit for innovative work going substantially beyond the minimal requirements in terms of evaluating alternative smoothing schemes.