

CS 224N / Ling 237

Programming Assignment 1: Language Modeling

Due Monday 16 April 2007

This assignment may be done individually or in groups of two. We strongly encourage collaboration, however your submission *must include a statement describing the contributions of each collaborator*. See the collaboration policy on the website (<http://cs224n.stanford.edu/assignments.html#collab>).

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 Leland machines (such as `elaine.stanford.edu`), make sure you can access the following directories:

```
/afs/ir/class/cs224n/pa1/java/ : the Java code provided for this course
/afs/ir/class/cs224n/pa1/data/  : the data sets used in this assignment
```

Copy the `pa1/java/` directory to your local directory and make sure you can compile the code without errors. The code compiles under JDK 1.5, which is the version installed on the Leland machines.

To ease compilation, we've 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 (and a symlink to the `ant` executable). Thus, to copy the source files and compile them with `ant`, you can use the following sequence of commands:

```
cd ~
mkdir -p cs224n/pa1
cd cs224n/pa1
cp -r /afs/ir/class/cs224n/pa1/java .
cd java
./ant
```

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/pa1/java/
mkdir classes/
javac -d classes src/**/*.java
```

Once you've compiled the code successfully, you need to make sure 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. Type `printenv CLASSPATH`. If nothing is printed, your CLASSPATH is empty and you can set it as follows:

```
setenv CLASSPATH ./classes
```

Otherwise, if something was printed out, enter the following to append to the variable:

```
setenv CLASSPATH ${CLASSPATH}:/classes
```

Now you're ready to run the test. From directory `~/cs224n/pa1/java/` enter:

```
java cs224n.assignments.LanguageModelTester
```

If everything's working, you'll get some output describing the construction and testing of a (rather crappy) language model. The next section will help you make sense of what you're seeing.

2 Using the LanguageModelTester

Take a look at the `main()` method of `LanguageModelTester.java`, and examine its output. This class has the job of managing data files and constructing and testing a language model. Its behavior is controlled via command-line options. Each command-line option has a default value, and the effective values are printed at the beginning of each run. You can use shell scripts to easily configure options for a run—we've supplied a shell script called `run` that will give you the idea.

The `-model` option specifies the fully qualified class name of a language model to be tested. Its default value is `cs224n.langmodel.EmpiricalUnigramLanguageModel`, a bare-bones language model implementation we've provided. Although this is a very poor language model, it illustrates the interface (`cs224n.langmodel.LanguageModel`) that you'll need to follow in implementing your own language models. A `LanguageModel` should implement a no-argument constructor, and must implement three other methods:

- `train(Collection<List<String>> trainingSentences)`. Trains the model from the supplied collection of training sentences. Note that these sentence collections are disk-backed, so doing anything other than iterating over them will be very slow, and should be avoided.
- `getWordProbability(List<String> sentence, int index)`. Returns the probability of the word at *index*, according to the model, within the specified sentence.
- `getSentenceProbability(List<String> sentence)`. Returns the probability, according to the model, of the specified sentence. Note that this method and the previous method should be consistent with one another, and in all likelihood this method will call that method.
- `generateSentence()`. Returns a random sentence sampled according to the model.

The `-data` option to `LanguageModelTester` specifies the directory in which to find data. By default, this is `/afs/ir/class/cs224n/pa1/data/`; if you copy data to your own machine, you'll want to override this option.

The `-train` and `-test` options specify the names of sentence files (containing one sentence per line) in the data directory to be used as training and test data. The default values are `treebank-train.sent.txt` and `treebank-test.sent.txt`. These files contain sentences from the Wall Street Journal

portion of the Penn Treebank. (For more details on the origin of the data, see the README files in the data directories.)

After loading the training and test sentences, the `LanguageModelTester` will create a language model of the specified class, and train it using the specified training sentences. It will then compute the perplexity of the test sentences with respect to the language model. Good perplexity numbers would be under 200; but when the supplied unigram model is trained on the Treebank data, it gets a perplexity of around 1500, which is very poor.

Next, if the `-hub` option is set to `true` (the default), the `LanguageModelTester` will do an evaluation using a set of HUB speech recognition problems. (“HUB” refers to a series of evaluations of speech recognition technology done by the National Institute of Standards and Technology (NIST) – see <http://www.nist.gov/speech/tests/>.) Each HUB problem contains 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.)

The tester first computes the perplexity of the HUB correct answers with respect to the language model. The HUB perplexity will in general be worse (larger) than the test set perplexity, since the HUB examples are drawn from a slightly different source, so our WSJ language models will be worse at predicting them. Note that language models most standardly treat all entirely unseen words as if they were a single UNKNOWN token (or event). This means that, for example, a good unigram model will actually assign a larger probability to an unknown word (“some unknown event”) than to a known but rare word.

As part of its HUB evaluation, the tester also computes a HUB word error rate (WER). The code takes the HUB 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).

Finally, if the `-generate` option is set to `true` (the default), the `LanguageModelTester` will print 10 random sentences generated according to the language model, so that you can get an intuitive sense of what the model is doing. Note that the outputs of the supplied unigram model aren’t even vaguely like well-formed English.

3 Your Job

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 validation data. 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. See also Chapter 6 of Manning and Schütze (1999).
5. Train on a larger data set. The standard Treebank data set contains only about 45,000 sentences, but we've also supplied a larger data set drawn from the BLLIP corpus which contains about 470,000 sentences.

A few programming tips:

- In the `cs224n.util` package there are a few utility classes that might be of use—particularly the `Counter` and `CounterMap` classes, which are helpful for counting n -grams.
- For development and debugging purposes, it can be helpful to work with a very tiny data set. For this purpose, we've supplied a data file called `mini.sent.txt` containing just 10 sentences. Of course, any results obtained from training or testing on this data are meaningless, but the run time will be very fast.
- If you're running out of memory, you can instruct Java to use more memory with the `-mx` option, for example `-mx300m`. You might also consider calling `String.intern()` on all new strings in order to conserve memory.

4 Submitting the assignment

4.1 The program: electronic submission

You will submit your program code using a Unix script that we've prepared. To submit your program, first put all the files to be submitted in one directory on a Leland machine (or any machine from which you can access the Leland AFS filesystem). This should include all source code files, but should not include compiled class files or large data files. Normally, your submission directory will have a subdirectory named `src` which contains all your source code. When you're ready to submit, type:

```
/afs/ir/class/cs224n/bin/submit-pa1
```

This will (recursively) copy everything in *your* submission directory into the official submission directory for the class. If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-pa1 -replace
```

We will compile and run your program on the Leland systems, using `ant` and our standard `build.xml` to compile, and using `java` to run. So, please make sure your program compiles and runs without difficulty on the Leland machines. If there's anything special we need to know about compiling or running your program, please include a `README` file with 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.

4.2 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.

Embedded written exercise: 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 .

Error analysis: The report should also include some error analysis. This means examining the outputs from your language model and looking for systematic errors. For the speech recognition task, this might involve identifying class of words, such as proper nouns, which are consistently misrecognized; for the sentence generation it might include identifying consistent weirdnesses, such as lack of agreement between subjects and verbs, in the randomly generated output. Note that these are just examples and we want you to look for all kinds of systematic errors. It may be tempting to wait until you have built all of your language models before you do your error analysis, but that would be a mistake. The point of error analysis is to find ways to improve your system, so it should be done frequently when developing your models. We would love to hear about classes of errors that you identified and fixed (and how you fixed them), as well as classes of errors still present in your models along with ideas for how they might be fixed.

5 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 this 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.

5.1 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.