1 Preliminaries

First, make sure you can access the course materials. The components are:

- code1.zip: the Java source code provided for this course, at http://grammar.ucsd.edu/courses/lign256/protected/code1.zip
- data1.zip: the data sets used in this assignment, at http://grammar.ucsd.edu/courses/lign256/protected/data1.zip

I have emailed out the username and password for accessing this code to everyone enrolled in the class. Please contact me if you haven’t received this information.

In addition, on your computer(s) you’ll need a version of the Java Development Kit (JDK), which you can get from here:

http://java.sun.com/javase/downloads/?intcmp=1281

Unzip the source files to your local working directory. Some of the classes and packages won’t be relevant until later assignments, but feel free to poke around. Make sure you can compile the entirety of the course code without errors (if you get warnings about unchecked casts, ignore them – that’s a Java 1.5 issue; if you cannot get the code to compile, please email me, stop by office hours, or post to the mailing list). If you are at the directory containing the src and classes directories, you can compile the provided code with

javac -d classes src/**//*.java src/**//*.java

You can then run a simple test file by typing

java -cp classes edu.berkeley.nlp.Test

---

1 This assignment is a minor adaptation of an assignment put together by Dan Klein. Many thanks to Dan for his permission to use this assignment, the code, and the datasets.
You should get a confirmation message back. You may wish to use an integrated development environment (IDE) such as Eclipse, available for free at [http://www.eclipse.org/downloads/](http://www.eclipse.org/downloads/) (this is recommended). If so, it is expected that you be able to set it up yourself. The majority of the students in the class are experienced in Java programming. If you are not, I highly recommend that you plan on collaborating with one of the more experienced students, to facilitate your own learning.

Next, unzip the data into a directory of your choice. For this assignment, the first Java file to inspect is:

```
src/edu/berkeley/nlp/assignments/LanguageModelTester.java
```

Try running it with:

```
java -cp classes edu.berkeley.nlp.assignments.LanguageModelTester
-path DATA -model baseline
```

where DATA is the directory containing the contents of the data zip.

If everything’s working, you’ll get some output about the performance of a baseline language model being tested. The code is reading in some newswire and building a basic unigram language model that I’ve provided. This is a phenomenally bad language model, as you can see from the strings it generates – you’ll improve on it.

## 2 Description

In this assignment, you will construct several language models and test them with the provided harness.

Take a look at the main method of LanguageModelTester.java, and its output.

**Preliminaries:** The language model you construct should be open-vocabulary. You can choose the vocabulary list, but see below under “Training” regarding what the word list should contain. Please state how you define your vocabulary list, and stick to a single list for the entire assignment.

**Training:** Several data objects are loaded by the harness. First, it loads about 1M words of WSJ text (from the Penn treebank, which will appear again later in class). These sentences have been ”speechified”, for example translating “$” to “dollars”, and tokenized for you. The WSJ data is split into training data (80%), validation (held-out) data (10%), and test data (10%). In addition to the WSJ text, the harness loads a set of speech recognition problems (from the HUB data set). Each HUB problem consists of a set of candidate transcriptions of a given spoken sentence. For this assignment, the candidate list always includes the correct transcription and never includes words seen less than twice in the WSJ training data. You should make sure that every word appearing in candidate list appears on your vocabulary list (hence your list must minimally consist of all words appearing at least twice in the training set, though you are welcome to use a larger list if you like). Each candidate transcription is accompanied by a pre-computed acoustic score, which represents
the degree to which an acoustic model matched that transcription. These lists are stored in SpeechNBestList objects. Once all the WSJ data and HUB lists are loaded, a language model is built from the WSJ training sentences (the validation sentences are ignored entirely by the provided baseline language model, but may be used by your implementations for tuning). For simplicity, all words are automatically converted to lower case (so don’t worry about this). Then, several tests are run using the resulting language model.

**Evaluation:** Each language model is tested in two ways. First, the harness calculates the perplexity of the WSJ test sentences. In the WSJ test data, there will be unknown words. Your language models should treat all entirely unseen words as if they were a single UNK 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 – this is because the aggregate probability of the UNK event is large, even though each specific unknown word itself may be rare. To emphasize, your model’s WSJ perplexity score will not strictly speaking be the perplexity of the exact test sentences, but the UNKed test sentences (a lower number).

Second, the harness will calculate the perplexity of the correct HUB transcriptions. This number will, in general, be worse than the WSJ perplexity, since these sentences are drawn from a different source. Language models predict less well on distributions which do not match their training data. The HUB sentences, however, will not contain any unseen words.

Third, the harness will compute a word error rate (WER) on the HUB recognition task. The code takes the candidate transcriptions, scores each one with the language model, and combines those scores with the pre-computed acoustic scores. 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: note that the candidates are only so bad to begin with (the lists are pre-pruned n-best lists). You should inspect the errors the system is making on the speech re-ranking task, by running the harness with the `-verbose` flag.

Finally, the harness will generating sentences by randomly sampling your language models. The provided language model’s outputs aren’t even vaguely like well-formed English, though yours will hopefully be a little better. Note that improved fluency of generation does not mean improved modeling of unseen sentences.

**Experiments:** You will implement at least two language models, though you can choose which specific ones to try out. To implement a language model, you need to do two things:

1. Write a class that implements the `edu.berkeley.nlp.langmodel.LanguageModel` interface; such a class needs two methods:
   - `getSentenceProbability(List<String> sentence)`, which returns the joint probability (not log-probability) of a sentence as a `double`;
   - `generateSentence()`, which returns a randomly generated sentence as a `List` of words, each of which is a `String`.

An example of how to implement this is the class
You might want to start by reading that class carefully to understand how it implements a unigram language model.

2. fill in the TODO and preceding lines in the

```
edu.berkeley.nlp.assignments.LanguageModelTester
```

class (the if statements with ‘‘your_model_here’’ in the test statement) such that passing -model XXX for your choice of XXX will cause your new language model(s) to be constructed.

The LanguageModelTester harness will do the rest for you. Along the way you must build the following:

- Either a Katz or Kneser-Ney estimator (bigram ok)
- A trigram (or higher order) model which uses an interpolation method of your choice
- A method which makes some use of the held-out data set

Note that if you build, for example, a Kneser-Ney trigram model with all hyperparameters tuned automatically on the held-out data, you’re technically done, though it will be more instructional to build up models of increasing complexity. While you are building your language models, it may be that lower perplexity, especially on the HUB sentences, will translate into a better WER, but don’t be surprised if it doesn’t. The actual performance of your systems does not directly impact your grade on this assignment, though I will announce students who do particularly interesting or effective things.

The most important determinant of your grade for the assignment is the degree to which you can present what you did clearly and make sense of what’s going on in your experiments using thoughtful error analysis. When you do see improvements in WER, where are they coming from, specifically? Try to localize the improvements as much as possible. Some example questions you might consider: Do the errors that are corrected by a given change to the language model make any sense? Are there changes to the models which substantially improve perplexity without improving WER? Do certain models generate better text? Why? Similarly, you should do some data analysis on the speech errors that you cannot correct. Are there cases where the language model isn’t selecting a candidate which seems clearly superior to a human reader? 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.
2.1 Collaboration

I highly encourage team collaboration in the programming portion of the assignment; however, each student should submit their own writeup. One way to collaborate might be to divide and conquer: each member of the collaboration has the job of implementing one choice of language model, the other members inspect their teammates’ work, and the team discusses the overall set of results together. There are other ways of doing collaboration, of course. Also be careful about collaborations that are too large: there is nothing inherently wrong with large collaborations (they allow you to cover a larger part of the model space!) but they can easily get unwieldy to organize and maintain.

2.2 Writeup

For this assignment, you should turn in a write-up of the work you’ve done, but not the code (it is sometimes useful to mention code choices or even snippets in write-ups, and this is fine). The write-up should specify what models you implemented and what significant choices you made. It should include tables or graphs of the perplexities, accuracies, etc., of your systems. It should also include some error analysis – enough to convince me that you looked at the specific behavior of your systems and thought about what it’s doing wrong and how you’d fix it. There is no set length for write-ups, but a ballpark length might be 3-4 pages, including your evaluation results, a graph or two, and some interesting examples. I’m more interested in knowing what observations you made about the models or data than having a reiteration of the formal definitions of the various models.

In the case of collaborations, the team should submit a brief statement of who did what work.

2.3 Miscellaneous advice

You are likely to build a model that requires you to allocate larger amounts of memory to Java than the default. To do this, pass the argument -mX<N>m to java (i.e., next to the -cp argument and before the class name), where <N> is the number of megabytes you wish to allocate.

In edu.berkeley.nlp.util there are some classes that might be of use – particularly the Counter and CounterMap classes. These make dealing with word to count and history to word to count maps much easier.

General advice regarding implementing models: it’s often said that laziness is a virtue in programming. One way of interpreting this maxim with respect to implementing models is don’t rush to implement a model that you don’t yet understand fully. I personally find that it (a) is ultimately more time-efficient, and (b) leads to better code, to carefully work out simple cases for your model on paper before starting to write code. The tiny-corpus Kneser-Ney model I showed on the board at the end of class Wednesday is an example of how this might be done for KN smoothing. You can even include these simple worked-out cases in your writeup (in fact, this is encouraged.) Make sure you understand the model.
completely! Otherwise you might find yourself retracing your steps all too often in the coding process.