cs294-5: Statistical Natural Language Processing
Assignment 1: Language Modeling

Due: Sept. 24th

Setup: On the instructional servers, make sure you can access the following directories under /home/ff/cs294-5:

java/src/edu/berkeley/nlp/ : the code provided for this course
corpora/assignment1 : the data sets used in this assignment

Copy the source files to your local java/src directory. Some of the files and directories won't be relevant until later assignments (and much of later code isn't present yet). 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). The important Java files to start out inspecting are:

assignments/LanguageModelTester.java
assignments/ProperNameTester.java

Try running the first one:

java edu.berkeley.nlp.assignments.LanguageModelTester /home/ff/cs294-5/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 I've provided (EmpiricalUnigramLanguageModel). This is 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 (edu.berkeley.nlp.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. 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; you can use this structure or not as you like.

Next, try to run the noun-phrase classifier:

java edu.berkeley.nlp.assignments.ProperNameTester /home/ff/cs294-5/corpora/assignment1

This trains and runs a trivial classifier (MostFrequentLabelClassifier) on a proper name classification task, and outputs accuracy figures. In this assignment, you'll build better language models and classifiers.

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 a SpeechNBestList object is created. These objects contain a correct answer and a set of candidate answers, along with acoustic scores for those candidates.

Once these objects are loaded, an astonishingly bas 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. Along the way you must build the following:

- A well-smoothed unigram model (using a Good-Turing estimator or a held-out estimator)
- A higher-order model (at least bigram) which uses an interpolation method of your choice
- Some method which makes some use of the held-out data set

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 does not directly impact your grade on this assignment.

What will impact your grade is 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). Similarly, you should do some 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.

Proper Name Identification: Now look at the main method of ProperNameTester.java. It loads training, held-out, and test sets, which are lists of LabeledDatum objects. These LabeledDatum objects each represent a proper noun phrase such as "Eastwood Park." Their features are the ordered lists of Characters making up the noun phrase string. Their labels are one of five strings: PLACE, MOVIE, DRUG, PERSON, or COMPANY. A classifier is then trained and tested. The trivial MostFrequentLabelClassifier always chooses MOVIE, since it is the most common label (barely).

You job here is to build two better classifiers (classes which implement ProbabilisticClassifier), as well as write some better evaluation code. Your classifiers should be a class-conditional language models:

- A class-conditional unigram model (a naïve-bayes classifier)
- A better class-conditional model (such as one based on a well-smoothed n-gram model)

You most certainly can reuse your language modeling code from the first part, and doing so will likely save a good deal of work. Again, the highest accuracy will win a title, but accuracy is only a secondary goal.

The primary goal is to understand what your classifiers are doing well, what they're doing badly, and why. Augment the provided evaluation code with two kinds of functionality. First, have it produce a confusion matrix, showing which label pairs are most often confused. Are the results surprising? Why do you think some pairs are easier or harder? Second, have your evaluation code investigate how your classifier's confidence correlates with accuracy. Are more confident (higher posterior likelihood) decisions more likely to be correct? Why or why not? Are there any individual errors that are particularly revealing as to why or how the system makes errors? Cases where you as a human would have trouble? If you were going to invest a large amount effort into raising the accuracy of this system, what would you do and why?

Write-ups: For this assignment, you should turn in a write-up of the work you've done, as well as the code. Your code doesn't have to be beautiful or glowing with comments, but I should be able to scan it and figure out what you did without too much pain. The write-up should specify what you built and what choices you made, and should include 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 3-6 pages, including your evaluation results, a graph or two, and some interesting examples.

Random Advice: In edu.berkeley.nlp.util there are 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.