Language Modeling and Enron Email Recovery

Programming Assignment 1
CS 224N / Ling 284

Due: 5pm January 19, 2011

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. 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 corn.stanford.edu or myth.stanford.edu) [1], 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 1.6, which is the version installed on most 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
```

Ant compiles all .java files in this directory structure, so you shouldn't have to change build.xml otherwise. 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

[1] see https://itservices.stanford.edu/service/unixcomputing for a list of Leland machines

```
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 (pretty bad) language model. The next section will help you make sense of what you're seeing.

2 The Code: 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.

LanguageModel Interface 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 four 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.
- checkModel(). Returns the sum of the probability distribution. A proper probability distribution should sum to 1. checkModel() will not be run if -check is set to false.
- 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.

Training and Testing 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 europarl-train.sent.txt and europarl-test.sent.txt. These files contain sentences from the Europarl corpus (For more details on the origin of the data, see the README files in the data directories). A second test set is also found in enron-test.sent.txt. This is a list of emails, described below. You need to test on both this and Europarl by changing the -test option.

After loading the training and test sentences, the LanguageModelTester will create a language model of the specified class (-model), and train it using the specified training sentences (-train). It will then compute the perplexity of the test sentences with respect to the language model. When the supplied unigram model is trained on the Europarl data, it gets a perplexity between 800 and 900, which is very poor. A reasonably good perplexity number should be around 200; a competitive perplexity can be around 100 on the test data.

Probability Distribution As part of the assignment you are required to ensure that all of your probability distributions are valid (sum to 1). To verify this experimentally, there is a checkModel method included in the LanguageModel prototype. An implementation of this appears in the EmpiricalUnigramModel which directly evaluates and returns $\sum_w P(w)$. When implementing checkModel for your bigram and trigram models, however, exhaustively evaluating $\sum_{w_2} P(w_2|w_1)$ for all w_1 will not be feasible. Thus you should check a reasonable number of random w_1's and ensure they individually sum to 1. If the -check flag is set to true, checkModel will be run on your current model and the returned number will be tested for proximity to 1. checkModel only needs to be run once on your final model, however, and disabling it will save you time on running repeated tests.

Model Generation 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.

Unknown Words A note about unknown words. Language models most standardly treat 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. Look for this in your error analysis during your experimentation.

3 The Task: Enron Email Recovery

It is late 2001 and the financial world has been rocked by irregularities in the accounting department of Enron Corporation. Investigators into the irregularities need your help. They have obtained by subpoena thousands of e-mails from its employees, but the e-mails were corrupted during the investigation and need to be recovered. All of the words in the emails are intact, but the word order has been randomly scrambled to varying degrees. Having taken CS224n, you suggest using a rich language model to help reconstruct the original word order. You decide to generate a list of possible sentences using the scrambled words, and choose the most likely sentence as determined by your language model.

Enron Data As a result of the Enron investigation (this is true), a large e-mail corpus was publicly released containing e-mails between many of its employees. We randomly selected several e-mail threads from this corpus and split them into sentences. For each sentence, we randomly scrambled it up to 100 times and output the scrambles into a file. You can find the data in

/afs/ir/class/cs224n/pa1/data/jumble. Each textN file is a single sentence's 100 scrambles. The original sentence is randomly placed in the list. You can find the correct answers in gold.

Running the Enron Test Thanks to your friendly CS224n staff, this Enron task is very easy to run. Simply set the -jumble option to true (the default), and the LanguageModelTester will automatically run everything for you. The code will call your model to score each scrambled sentence and return the sentence assigned the most probability. It then compares your best sentence to the gold original sentence and computes two scores:

- **% correct** This is the number of sentences that you matched exactly right, divided by the total number of sentences.
- **Word Error Rate** (WER) This is a measure of the distance your chosen sentence is from the original sentence. It is the minimum number of 'edits' you have to make to change your sentence into the correct one.

The tester will also compute the perplexity of the Enron correct emails with respect to the language model. This gives you a third indicator of how well your language model is performing. Note that the perplexity will in general be worse (larger) than the Europarl train/test set perplexity because the Enron text is a different domain than Europarl. The style of language and word use in emails is different from that in European Parliamant text. The former is more informal and will obviously discuss different topics than the latter.

Finally, there is a -showguesses option that, when set to true, will print out the top sentences as scored by your language model. It lets you view the actual Enron emails as reordered by your model, and can help in error analysis.

4 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 (on europarl-test.sent.txt and enron-test.sent.txt), WER on the word jumble, and in terms of the generated sentences looking much better (however, you should concentrate on improving your perplexity. If the other Enron metrics behave contrary to your expectations, however, we would like you to explore why that is).

Minimally, you should do the following:

1. Build 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. Experiment with 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. Implement a smoothing method which makes some use of validation data. For instance, it could set weighting parameters in a linear interpolation.
4. Implement checkModel() correctly. Each language model you build should define a proper probability distribution, and you should show in the writeup (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.
5. For each of your models, train with Europarl training data and record performance on (1) europarl-test.sent.txt perplexity, (2) enron-test.sent.txt perplexity, and (3) Enron WER and % correct.

6. Perform and report on error analysis between your implemented language models, and why you built them to solve specific problems that you observed. (more in section 4.1)

Some other things you might try:

7. 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).

8. Experiment with different training data. We only used 50,000 sentences from the Europarl data for the train/validate/test sets. There are 1,090,475 sentences unused, which you can find in data/europarl.lowercased.short.notused. For details, see data/README. In addition, we've included more Enron data in data/enron-train.sent.txt. How does training on Enron vs. Europarl perform on the different tests?

   Note: Give yourself enough time if you intend to run these tests. Depending on your implementation, these can take a fair amount of time to complete. Moreover, the JVM's memory requirements will also increase so running these tests on one of the clusters will probably be a good option.

4.1 Evaluation criteria

While you are building your language models, hopefully lower perplexity will translate into better % correct and WER performance on the Enron task, but don't be surprised if it doesn't. A best language-modeler title and a highest % correct 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 suggests you are doing something right, and bad performance something wrong).

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.
- Clear presentation of the language modeling methods you used.
- Clear presentation of each language model's results.
- 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 % correct and 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 improvements. (This will almost certainly require altering the testing code, and this is strongly encouraged.)
- **Error analysis** on your final and intermediate results. This is the most important part of your report. 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? What specific examples motivated your language model choices? The bottom line is that your write-up should include concrete examples of errors or error fixes, along with commentary. For the Enron task, this might involve identifying classes of words which are consistently misrecognized, such as proper nouns; for the sentence generation it might include identifying consistent anomalies, such as lack of agreement between subjects and verbs. 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 errors that you identified and fixed, as well as classes of errors still present in your models with ideas for how they might be fixed.

Note: Motivations for moving to higher-order n-gram models and various kinds of smoothing techniques were discussed in class. However, simply re-iterating these reasons will not get you full credit. We do expect in-depth analysis and examples or reasoning of why changing models improved (or worsened) performance.

4.2 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.

5 Submitting the assignment

5.1 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. 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. Your code doesn't have to be beautiful but we should be able to figure out what you did without too much pain.

5.2 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 6 pages, including your evaluation results, a graph or two, error analysis, 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.

5.3 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. Partial credit will be given for evaluating the impact of different training data.